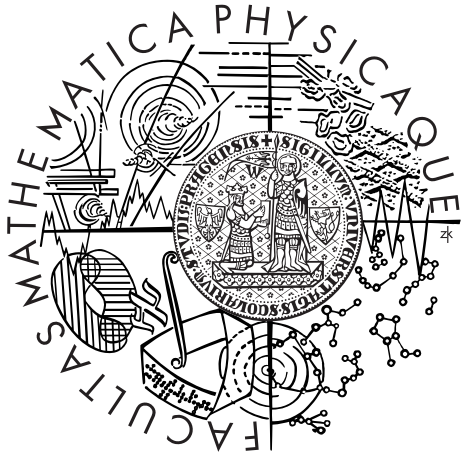
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Pavel Veselý

Artificial intelligence in abstract 2-player games

Department of Applied Mathematics

Supervisor of the bachelor thesis: RNDr. Tomáš Valla
Study programme: Computer Science
Specialization: General Computer Science

Prague 2012

I would like to thank my advisor Tomáš Valla for showing me the Project GIPF, particularly Tzaar, and for weekly meetings where we discussed work on the robot for Tzaar. Jitka Novotná and Pavel Dvořák were also on this meetings and they helped me with some fresh ideas. Petr Baudiš and Jan Hric had a seminar about game algorithms during the winter semester 2011/2012 where I learned about state-of-art algorithms for game playing programs.

I also would like to thank the Department of Applied Mathematics for a place and computation time on a server on which the robot for playing Tzaar lives and on which experiments took place. The robot plays with people on the french game server Boiteajeux.net despite the fact that robots are not "officially supported" there. Some players on this server try to defeat the robot, a few of them more than several times, and thus they are giving me a valuable feedback on the robot's playing.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date ..... Pavel Veselý

Název práce: Umělá inteligence v abstraktních hrách dvou hráčů

Autor: Pavel Veselý

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: RNDr. Tomáš Valla, Informatický ústav Univerzity Karlovy

Abstrakt: V této práci se zaměříme na algoritmy pro hledání nejlepšího tahu v zadané pozici abstraktní strategické hry dvou hráčů. Popíšeme algoritmy Alfa-beta a Proof-number search včetně jejich vylepšení a přispějeme novými nápady na jejich zrychlení. Rovněž navrhneme postup, jak vybírat náhodně mezi tahy ne o moc horšími než nejlepší nalezený a jak hrát v prohraných pozicích. Algoritmy nasadíme na hru Tzaar, která je zvláštní velkým počtem možných tahů, což ji dělá obtížnou pro počítač. Naším cílem je vytvořit co nejlepšího robota na hraní Tzaaru. Ukážeme, že naše umělá inteligence dokáže hrát na úrovni nejlepších lidských i počítačových hráčů na internetu. Také na základě experimentů rozebereme, jak jednotlivá vylepšení algoritmů pomáhají v zrychlení výpočtů u této hry.

Klíčová slova: herní stromy, Minimax, Alfa-beta, Proof-number search, Tzaar

Title: Artificial intelligence in abstract 2-player games

Author: Pavel Veselý

Department: Department of Applied Mathematics

Supervisor: RNDr. Tomáš Valla, Computer Science Institute of Charles University

Abstract: In this thesis we focus on algorithms for searching for the best move in a given position in an abstract strategy 2-player game. We describe algorithms Alpha-beta and Proof-number Search with their enhancements and we come with new ideas for making them quicker. We also propose an algorithm for choosing randomly between moves not much worse than the best move and ideas how to play in lost positions. We apply the algorithms on the game Tzaar which is special for having a lot of possible moves which makes the game hard for a computer. Our goal is to create a robot for playing Tzaar as good as possible. We show that our artificial intelligence can play on the level of best human and computer players. We also examine experimentally how enhancements of the algorithms help making computations quicker in this game.

Keywords: Game Trees, Minimax, Alpha-beta, Proof-number Search, Tzaar

Contents

Introduction

From the beginning of the computer era people are working on game playing programs. The first challenge was to create a robot that can win against the top human in Chess. In 1996 Garry Kasparov, being currently the best Chess player, lost a game against the computer Deep Blue, but still won whole match consisting of six games. Nowadays the strength of best computer programs for Chess is much higher than the strength of best people.

In the meantime, algorithms in Chess programs were adapted to other games. They were not always successful against professional players, for example in the game Go, best human players are still much better than any program. For these games new algorithms were invented and the progress in this field is still going on.

The game Tzaar, originally written as TZAAR, is very new in comparison with Chess and Go. It was created in 2007 by Kris Brum as the sixth game of the Project GIPF in which there are six abstract strategy two-player games on a hexagonal grid (not always the same). Some computer programs for Tzaar are available, but nobody has done a research on the game properties of Tzaar and on algorithms appropriate for Tzaar (up to my best knowledge). The game GIPF, the first in the Project GIPF, has already been studied, see [19].

Tzaar has one special property that makes it very difficult for computers. A player can move with stones on the board twice in one turn which leads to a few thousands of different choices how to play in a position in the beginning of the game. This number is at least ten times more than the number of possible moves in Go, but Tzaar games are quite short (typically up to 28 plies, i.e., turns of a player).

The goal of this thesis is to create a strong Tzaar artificial intelligence (AI) that is able to play well with advanced players. To reach the goal we use algorithms Alpha-beta and Proof-number Search modified with many enhancements. Both algorithms work on a *game tree* in which every node corresponds to a position and the root node is the position for that we want to find the best move. A node N is a son of a node P if and only if there is a move from the position in the node P to the position corresponding to the node N. Note that one position can be more than once in the tree.

Large number of enhancements for Alpha-beta was invented, so we pick only the most important and appropriate for Tzaar. Proof-number Search is newer algorithm than Alpha-beta and we use most of state-of-art enhancements that are dealing with problems occurring in Tzaar.

We also propose some new ideas for the algorithms. Our robot that we created is intended to play against people, thus some randomness in the beginning of every game is needed, otherwise one can find a strategy how to win against the robot. We give a new simple method how to select a random move that is not much worse than the best move using Alpha-beta algorithm.

We also propose an improvement for the Weak Proof-number Search [5] called the Heuristic Weak Proof-number Search using an idea mentioned briefly by Kishimoto [9]. We also show how to estimate the number of nodes of the game tree that can Proof-number Search visit in a given time limit.

It happens quite often that the robot finds itself in a lost position. We bring some ideas for both Alpha-beta and the Proof-number Search how to play reasonably in such positions.

In the first chapter we present the game Tzaar together with its properties, strategies and existing robots for automatic playing Tzaar. Chapter 2 covers appropriate algorithms for Tzaar in general way, namely Alpha-beta and the Proof-number Search with their enhancements, and the next chapter describes their implementation in the domain of Tzaar with details of Tzaar specific heuristics. In the fourth chapter we test the algorithms experimentally, show how enhancements and parameters of the algorithms help making the search quicker, and how is our robot successful against other programs and people.

1. Tzaar Game

In this chapter we introduce rules of the game Tzaar and discuss its game properties. Popularity of Tzaar and some strategic notes on playing are mentioned. We also take a look at other programs for playing Tzaar against a computer.

1.1 Tzaar Rules

Tzaar is an abstract strategy two-player game quite similar to Chess and Go. Abstract means that the game does not have a theme. It also has perfect information, i.e., both players know all information about the current position in the game, and there is no randomness. White player and black player take turns, white has the first turn.

The board for Tzaar is hexagonal and consists of 30 lines that makes 60 intersections. There is no intersection in the middle of the board. In the starting position there are 60 pieces (stones), 30 of them are white and 30 are black. Pieces have three types, six pieces of each color are Tzaars, nine are Tzarras and 15 are Totts, see Figure 1.1. The stones are on the intersections of the lines.

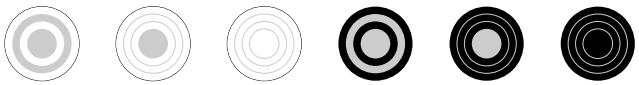


Figure 1.1: Tzaar pieces

The starting position has one stone on each intersection. The placement could be random or players can use a fixed starting position which is defined in official rules [37] and shown on Figure 1.2.

Pieces can make *stacks*, that means, towers of pieces of the same color. In the beginning, all stacks on the board have height one. A stack is one entity, thus it cannot be divided into two stacks.

Each player's turn consists of two moves with stones. The first move must be a *capture*. The player on turn takes one of his stacks and moves it on an intersection where his opponent has a stack. The move is done in the direction of a line on which is the moved stack. A piece cannot jump over other pieces and over the middle of the board, only over arbitrary number of empty intersections (spaces). Players can move only with pieces of their color and no piece can jump on an empty intersection. Captured stack must have height at most the height of capturing stack. Captured pieces go out of the board.

The second move of a turn can be another capture move, or a stacking move, or a pass move. *Passing* means that player on turn does not move with any piece, so nothing changes and the other player is on turn. *Stacking move* is done similarly to a capture move, but the player jumps with a stack on a stack of his own color. This move makes a stack of height equal to the sum of heights of the stack which jumped and the stack on which was jumped. There is no restriction on heights and types of stacks, for example a Tzaar stack with height 1 can jump on a Tott stack with height 8.

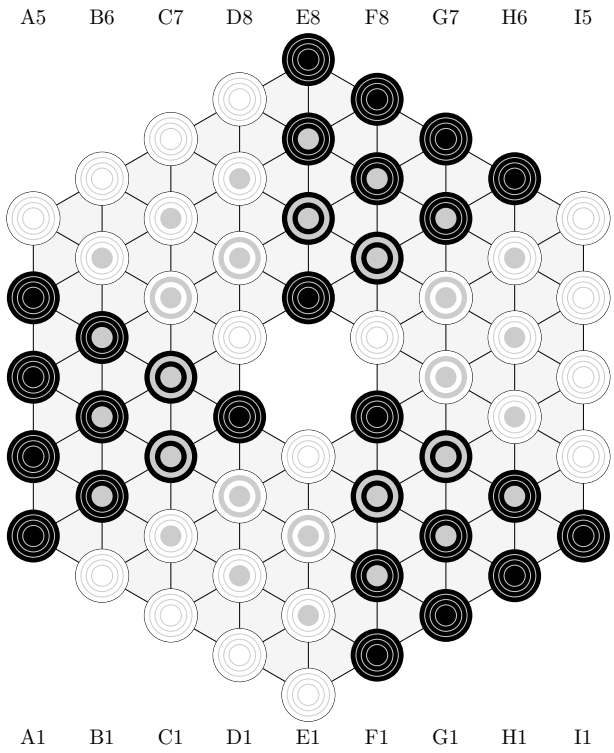


Figure 1.2: Fixed starting position

The first turn of white player consists only of a capture, then there are two moves per turn and the first move must be a capture.

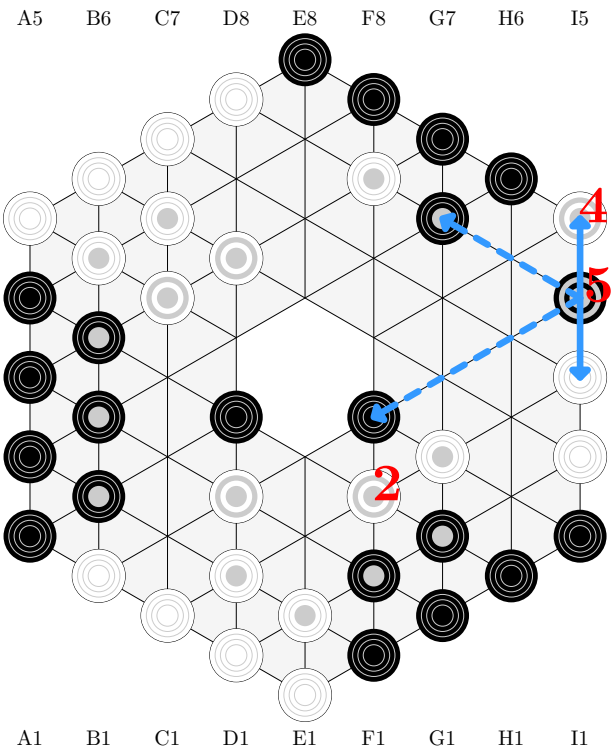


Figure 1.3: Example of possible moves of the black Tzaar stack in the second move of a turn. The dashed arrows represent stacking moves and red numbers mean the height of a stack when it is more than one.

A player loses when he has no stack of any of three types, or if he is on turn and he has no possible capture for the first move (but he can make his last possible capture and then pass or stack). Only visible stones, i.e., stones on the top of stacks, count for the presence of three types.

A draw is not possible in Tzaar. Quite surprisingly, according to rules, a player can "commit suicide" (lose) by stacking his piece on the last stack of a stone type.

Tournament version of the game begins with an empty board. The players take turns in which they put a piece on an empty intersection on the board. The order of placing the stones is arbitrary. After putting all stones on the board, all intersections are occupied and the game starts like an usual game. In this thesis we deal mostly with games without this placement phase, i.e., not in the tournament version.

1.2 Tzaar History

Tzaar, originally written as TZAAR, is a very new game. It was invented by Kris Brum and published in 2007. It is a part of the Project GIPF, a set of six abstract strategy two-player games. Tzaar replaced TAMSK, the second game of the project, and no other game was added to the project after Tzaar, since it has now six games as the author intended.

Tzaar has won quite a lot of awards. The most important is probably Games Magazine's award Game of the Year 2009 [26]. In 2008 it has been honoured Spiel des Jahres Recommendation [35] and nominated to International Gamers Award – General Strategy: Two-players [31], Golden Geek Best 2-Player Board Game [28] and Nederlandse Spellenprijs [32].

It is also high rated on web, for example on BoardGameGeek.com where it has the second highest rating between abstract games (more than 1 000 users have voted up to July 13, 2012) [22].

1.3 Strategies

In this section we discuss some strategies how to play Tzaar. The strategies are based on playing with human and computer opponents (also other than robot described in this text) and there are some positions where they do not hold. The official Tzaar page has some strategy tips too [38].

In the second move of a turn a player, has three possibilities: capture, stack or pass. In most situations the stacking is the most reasonable, because it makes one of stacks more powerful and more safe against opponent's stacks. The other reason is that the opponent loses capturing possibilities, thus he would more likely run out of captures and lose in the endgame.

Capturing again (so called *double capture move*) is appropriate if the opponent is running out of stones of a type (he should not have high stack of that type) or if a high stack can be captured – height had better be more than two, because by capturing stacks of size two, one can lose capturing possibilities. More importantly double capturing two pieces with height only one leads mostly to losing capturing possibilities.

Passing move does not occur in many games. It is worth doing only in the endgame when stacking is not possible or would result in loss. See Figure 1.3 for an example of such a position.

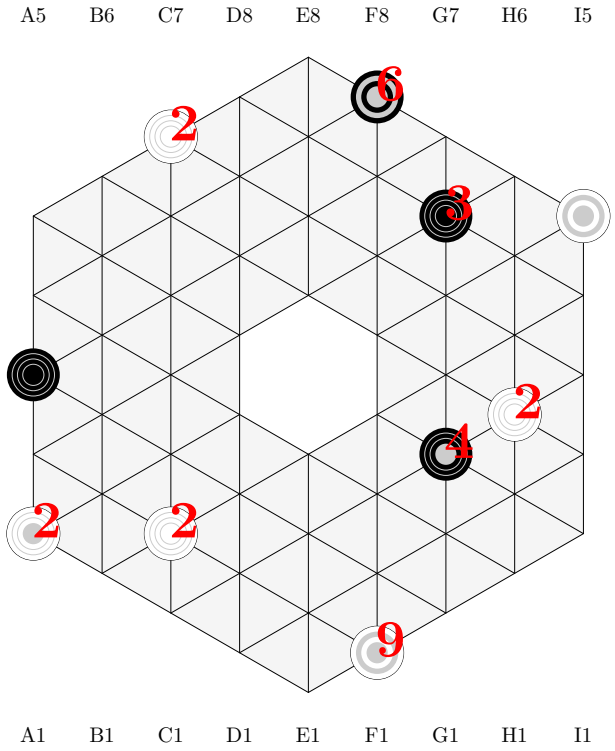


Figure 1.4: In this position black is on turn. After the last black Tzaar stack captures white Tzaar piece in the right corner, the only move not leading to a loss for black is the pass move.

We may also observe that black player has a little advantage, because he is stacking first and thus he can often threaten his opponent by chasing his stack.

The first move is always a capture move, but it often depends which type of stone is captured. It is mostly the best to capture a type of stone from that the opponent does not have a high stack. In a typical game a player starts creating a stack of Tzaar, thus it is probably convenient to capture Tzarras.

There are generally two strategies how to stack. The first is creating one high stack that is powerful and that can capture all opponent's stacks or that forces opponent to raise his stack although it is a little higher. The second strategy is based on creating more lower stacks, mostly of size two, but it is safer when some of them have height three.

It is not known to me which is better, probably the first for black player and the second for white player. With the first strategy a player can quite easily threaten or even capture small opponent's stacks, but with the second it is sometimes impossible for the opponent to create a new stack (it would be captured immediately) and the opponent can lose because of it. The second strategy is more reasonable in the endgame, since it decreases the opponent's capturing possibilities. Having a stack much higher than all other opponent's stacks is also only a little advantageous.

These strategy observations were mostly about material; now we discuss some positional strategy features. One important positional advantage is that high

stacks can move easily to any direction and thus they threaten large part of board. We can conclude that it is better to have high stacks in the middle of the board, not on the border. Moreover from the middle of the board a stack can nearly always escape from a threat by a higher stack. The worst position is in one of the six corners. It is also good to limit possibilities to move for opponent's high stacks.

When one stack is isolated, i.e., it cannot be captured and it cannot capture or stack, because there are no other pieces on the same lines as the isolated stack, the type of that stone is safe. Hence the player cannot run out of it which is a great advantage. But isolating high stack is not good, because then the stack cannot be used for capturing opponent's stacks. As the game is coming to the end it is useful for a player to limit opponent's possibilities of captures and also prepare own capture possibilities.

Moving pieces (stacks of height one) from the middle of the board is less important and useful only before the endgame – otherwise the opponent can jump through them and maybe double-capture a high stack.

1.4 Game Properties of Tzaar

Before creating a computer AI for playing Tzaar, it is convenient to know game properties. According to Heule and Rothkrantz [7], and Allis [1], we try to estimate the state space and game tree complexity of Tzaar. Then we show some other properties of Tzaar, e.g., branching factor in different parts of the game, the number of starting positions, and the number of positions in the endgame with a fixed number of stacks.

First we look at the *maximum height of a stack*. Before each stacking move of a player his opponent must capture him a piece (at least one capture before stacking to height two, at least two captures before height three...). There should be two stones of another types visible and there are 30 pieces of each color, so the maximum stack height is 14 because of 13 captures before stacking to the height 14 and two other stones visible.

An upper bound on the number of different legal game positions obtainable from any initial position, i.e., *the state space complexity*, is the sum of the number of positions with fixed number of stacks on the board. With v free fields on the board, there are $\binom{60}{v}$ different choices of fields for stacks. Let the variable k denote the number of stones of white color on the board (including the ones inside a stack). The upper bound on k can be obtained by the number of necessary captures before v free fields are on the board. The variable s stands for the number of white stacks.

For black player the number of stones is in the variable l and the number of black stacks is $60 - v - s$. The binomial coefficient $\binom{k-1}{s-1}$ means the number of different stack heights for s stacks with k white pieces; the number of different choices of fields for white pieces is $\binom{60-v}{s}$ and 3^s is the number of different types of white stacks. Similar formulas holds for black player. We note that two positions that differ only in types of stones inside stacks (not on the top) are the same for us. These observations give us a formula for the upper bound on the number of possible states:

$$\sum_{v=1}^{55}\binom{60}{v}\cdot\sum_{k=2}^{30-\lfloor\frac{1}{4}v\rfloor}\sum_{s=2}^{\min(k,58-v)}\binom{60-v}{s}\binom{k-1}{s-1}\cdot 3^s\cdot\sum_{l=60-v-s}^{30-\lfloor\frac{1}{4}v\rfloor}\binom{l-1}{59-v-s}\cdot 3^{60-s-v}$$

$$\doteq 9.17\cdot 10^{57}$$

We did not take symmetries into account. The position can be rotated by 60°, 120°, 180°, 240° and 270° and still it is nearly the same for both players, i.e., the advantage of white player is not changed by a rotation. We can also mirror the position by one of three axes between opposite corners of the board or by one of three axes between centres of opposite sides. Mirroring twice by two axes (it does not matter whether they are between the corners or the centres of sides) results in a rotated position, thus there are 12 positions that are the same due to the rotation and reflection symmetries. This decreases our upper bound of the state space complexity to $7.64\cdot 10^{56}$.

We can observe that this is the number of positions which can be reached from all starting positions altogether, but some positions (and maybe most of them) can be obtained from more than one initial position. Note that positions in the placement phase of the tournament version are already counted.

The *game tree complexity* is defined as the number of leaf nodes in the minimum solution tree of the initial position(s) of the game and usually crudely counted as the average branching factor (according to the depth) to the power of the average game length in plies, i.e., turns of a player. Since pass moves are played rarely we can estimate that in every turn of a player, except for the first turn of white player, two stacks disappear from the board – one captured and one captured, or stacked. There should be at least 5 stacks in the final position, thus the upper bound on the number of plies in a game where players do not pass is 28.

The *branching factor*, i.e., the number of possibilities how a player can play in one turn, depends on the starting position. The fixed starting position has the maximum branching factor around 6 300, but there are starting positions with the branching factor up to 14 000. From the beginning of the game, the branching factor is decreasing as the stones are captured or stacked. See Figure 1.5 for maximum, average and minimum branching factor according to the number of stacks on the board (data were taken from more than 5 000 positions that occurred during games on Boiteajeux.net).

If we multiply average branching factors, we get approximately 10^{81} which is an estimate of the game tree complexity.

In the tournament version of the game, the search tree is even much more larger because of the placement phase. The first player has 60 possibilities where to put his first piece, the second player has 59 possibilities, first then has 58 ... Thus the game tree of the placement phase has 60! leaves (without taking symmetries into account) and the game tree complexity of the tournament version of the game is $60!\cdot 10^{81}\doteq 10^{163}$.

The number of different starting positions is the number of permutations with repetitions:

$$\frac{60!}{(15!\cdot 9!\cdot 6!)^2}\doteq 7.13\cdot 10^{40}$$

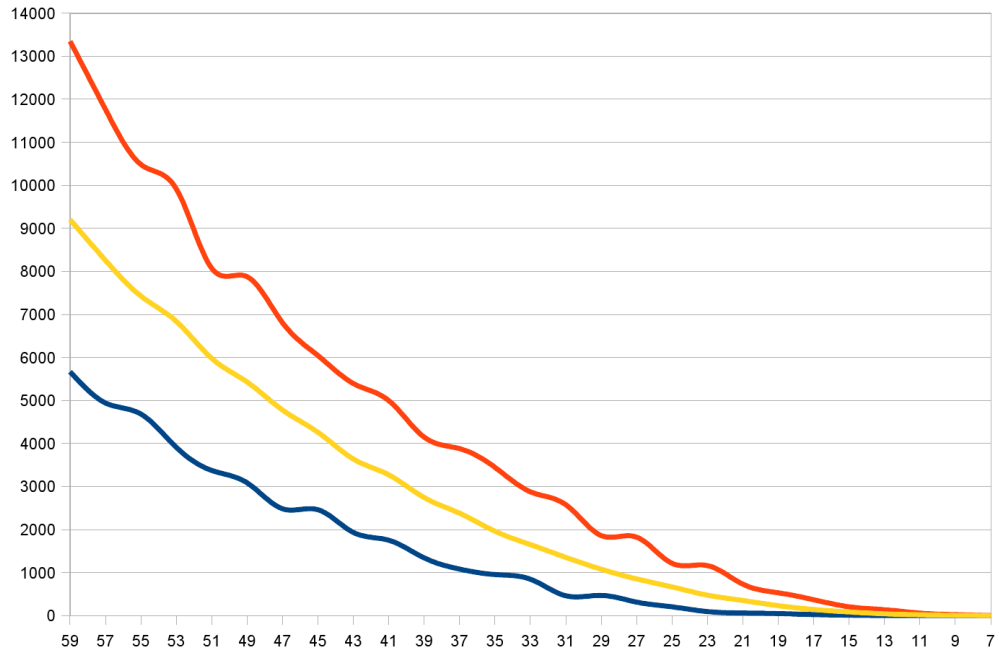


Figure 1.5: Maximum (red), average (yellow) and minimum (blue) branching factor according to the number of stacks on the board.

There are again 12 symmetries which result in $5.94 \cdot 10^{39}$ different starting positions.

It is also handy to know the number of endgame positions. We count the number of positions with six stacks of different types or colors — if there are two stones of the same color and type, the position is won by one of players. We observe that the number of positions with more than six stacks is higher. The number of positions with exactly six stacks is the number of different choices of six fields on the board multiplied by the number of permutations of six stacks and the number of different stack sizes for each stone type and for each player. The maximum sum of stack sizes for a player is 16, because there should be a capture before each stacking. These observations give us the following formula:

$$\binom{60}{6} \cdot 6! \cdot \left(\sum_{i=3}^{16} \binom{i-1}{2} \right)^2 = 1.13 \cdot 10^{16}$$

where i is the sum of stack heights for one player.

After taking symmetries into account, we get $9.42 \cdot 10^{14}$ different positions with six different stacks.

1.5 Existing Tzaar Artificial Intelligences

Up to June 2012 there are these Tzaar programs with an artificial intelligence (AI) available on the Internet: hsTzaar [30], TZ1 [36], students' and teacher's works at the Department of Mathematical Sciences at the University of Alaska [39] and robots on BoardSpace.net [23]. I tried to play with them — for an idea of my strength in playing Tzaar, my ELO rating on Boiteajeux.net [24] is 2 173, the 6th

highest (up to July 30, 2012). The starting ELO is 1 500 and there are more than 50 players with ELO over 1 800.

The *hsTzaar* program [30] published as an open source project is created in Haskell. It is developed from the older program *htzaar*. It offers graphical interface and saving and loading games. There are four levels of an AI opponent. The second best plays quite reasonable moves, but I can beat it in nearly every game. The best AI opponent is very slow (in the beginning it thinks more than a few minutes in each move) and also beatable. The first two levels are very quick, but still play good although I can defeat them quite easily.

TZ1 [36] is written in Java and also has a graphical interface. The AI opponent plays very bad moves and can be beaten by intermediate players easily.

There are four available students and teachers works on University of Alaska's site [39]: *Mockinator*, *Mockinator++* (improved Mockinator), *BiTzaarBot* (with algorithm inspired by the Monte-Carlo Tree Search) and *GreensteinTzaarAI*. They are able to connect to the Daedalus Game Manager [25] which has graphic user interface for Tzaar and is able to maintain games between AIs. GreensteinTzaarAI won a tournament between these programs [39].

Robots on BoardSpace.net [23] run under Java. The Dumbot is the weakest robot there, but Smartbot and Bestbot also do not play well and experienced players are able to win against them nearly every time. They even do not stack the first time they can. Additionally Bestbot can think in some positions for more than half an hour (this also depends on the hardware).

For a comparison between these programs and AI described in this thesis see Section 4.2.2.

2. Search Algorithms for Board Games

This chapter describes existing algorithms for searching for the best move in two-player strategic games without randomness and with complete information, e.g., Chess, Go and Tzaar. In these games, one theoretical method for searching the best move is generating whole game tree from a given position. We want to find a winning strategy, that is, a move resulting in a position from which all opponent's moves leads to a winning position for us, i.e., for each such position there exists our move leading to a position from which all opponent's moves lead to a winning position for us (with recursively the same definition).

This strategy can be found (if there is any) by *Minimax algorithm* that we describe in Section 2.1.1. So searching the whole game tree would result in the best possible play in winning positions (the move leading to a win is always found), but the problem is that game trees for most board games including Chess, Go and Tzaar are too large to be searched and only a very small part of the game tree can be searched within a given time, that is, up to several minutes.

We describe basically two different algorithms, Alpha-beta and Proof-number Search, together with some of their enhancements. Note that there are other algorithms, like Dependency-based search [1] and Lambda search [18], but they are not suitable in the domain of Tzaar as shown in Section 3.2. Monte Carlo Tree Search [2, 11] may be a good choice for Tzaar, but it was not tried.

2.1 Alpha-beta Algorithm

Alpha-beta algorithm is an extension of the Minimax algorithm by pruning non perspective branches of the game tree, so we first describe Minimax and then Alpha-beta. There are many enhancements for Alpha-beta, most of them are for making pruning as effective as possible. We describe the most common and appropriate for Tzaar.

2.1.1 Minimax

The base of the Minimax algorithm is a depth-first search on the game tree in which the root node is the position for which we want to find the best move. Because of a time limit the tree is searched only to a given depth. Part of the game tree that is searched is called a *search tree*.

A *value* is assigned to each node in the search tree and node evaluation is done recursively. We sometimes use a *value of a move* that means simply a value of the position after executing the move. There are two players which we name *Max* and *Min* and let Max be on turn in the root position of the tree. Max wants the value of his nodes to be as high as possible and Min as low as possible.

Leaves are evaluated directly, final positions in which the game ends obtain $+\infty$ if won by Max, $-\infty$ if won by Min, and zero in the case of a tie. Other leaf positions have value counted by a heuristic *evaluation function* that returns value higher than zero in positions better for Max player, approximately 0 in balanced

positions and less than zero in positions in which Min has an advantage. It should hold that the better position for Max, the higher the value is and vice versa for Min. An evaluation function for Tzaar is described in Section 3.5.

Nodes inside the tree including the root node count the value from values of their sons in the tree. When Max is on turn, a node obtains maximum of values of his sons, for Min it is minimum. The best move from the root position leads to the son that has the same value as the root.

Pseudocode of Minimax is in Algorithm 1. The function returns the best possible value and move for the player on turn. Note that ∞ is a constant that should be more than any value which the evaluation function can return.

Algorithm 1 Minimax algorithm

```
 1: function MINIMAX(position, depth, onTurn)
 2:     if depth = 0 or endOfGame(position) then                ▷ leaf node
 3:         return (evaluate(position), emptyMove)
 4:     end if
 5:     if onTurn = Max then
 6:         bestValue ← −∞ − 1
 7:         bestMove ← emptyMove                    ▷ Try all moves of player Max
 8:         for all m in generateMoves(position, Max) do:
 9:             (value, move) ← minimax(executeMove(m), depth − 1, Min)
10:             if value > bestValue then
11:                 bestValue ← value
12:                 bestMove ← m
13:             end if
14:         end for
15:         return (bestValue, bestMove)
16:     end if
17:     if onTurn = Min then
18:         bestValue ← ∞ + 1
19:         bestMove ← emptyMove                    ▷ Try all moves of player Min
20:         for all m in generateMoves(position, Min) do:
21:             (value, move) ← minimax(executeMove(m), depth − 1, Max)
22:             if value < bestValue then
23:                 bestValue ← value
24:                 bestMove ← m
25:             end if
26:         end for
27:         return (bestValue, bestMove)
28:     end if
29: end function
```

In this Minimax implementation there should be two different counting of the value in each internal node, one for each player. To have only one, *Negamax* adjustment is used. A player on turn is always Max, i.e., the evaluation function returns high values when the player on turn has an advantage and the maximum of values of sons is always counted. Between search tree levels the value is multiplied by -1.

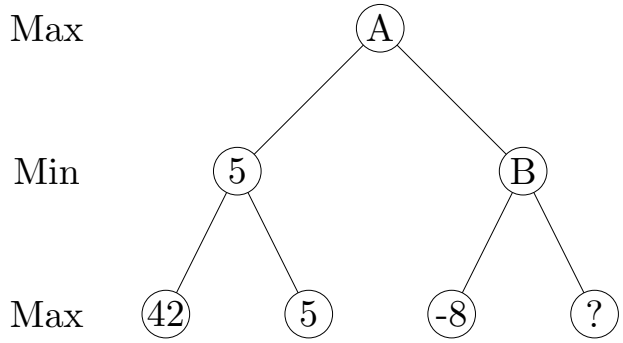


Figure 2.1: Example of a search tree. The position ? need not to be searched, since the position B has value at most -8 and thus the root position have value 5.

The time complexity of this algorithm is $\mathcal{O}(b^d)$ where b is the average branching factor and d is the search depth. The space complexity is $\mathcal{O}(d)$ because of a stack for remembering nodes on the path to the leaves.

2.1.2 Alpha-beta Pruning

The *Alpha-beta pruning* (or the Alpha-beta algorithm) is an important improvement of the Minimax algorithm. It is based on the observation that mostly not all nodes have to be searched to obtain a value of the root position. For example see a search tree on Figure 2.1.

This observation can be generalized. While searching the tree we remember variables *alpha*, the best achievable value for player Max, and *beta*, the best achievable value for Min. The best achievable means that whatever opponent moves are, the value is guaranteed – it can be better for the player, but not worse. At the beginning *alpha* is $-\infty$ (Max can lose the game) and *beta* is $+\infty$ (Min can lose too).

In the position where Max is on turn, variable *alpha* is updated and if it exceeds *beta*, searching the subtree of the node is interrupted. This interruption is called *cutoff*. The similar with *alpha* and *beta* swapped holds for positions of Min. Thus always *alpha* is strictly less than *beta*.

In Algorithm 2 we present a pseudocode of Alpha-beta adjusted in the way of Negamax – we count always maximum, values are multiplied by -1 between levels, and additionally *alpha* and *beta* are swapped. Note that the function is returning only a value for simplicity.

Alpha-beta can run in the worst case in the same time as Minimax – when nodes are examined from the worst to the best. But if they are searched from the best to the worst, there is the maximum number of prunes and in every second level of the search tree (odd or even depending on who has the advantage) only one move has to be examined. Hence the time complexity of the algorithm is $\mathcal{O}(b^d)$ and $\Omega(b^{d/2})$. For more elaborated analysis of the algorithm and proof of its correctness see the paper of Knuth and Moore [10].

Next we describe how to choose randomly between moves that are only a little worse than the best move and then some enhancements for the Alpha-beta algorithm. See Section 4.1.1 how the enhancements have effect on the duration of the search in the domain of Tzaar.

Algorithm 2 Alpha-beta algorithm

```
1: function alphaBeta(position, depth, onTurn, alpha, beta)
2:     if depth = 0 or endOfGame(position) then                ▷ leaf node
3:         return evaluate(position)
4:     end if
5:     bestValue ← −∞ − 1
6:     for all m in generateMoves(position, onTurn) do:        ▷ Try all moves
7:         value ← alphaBeta(executeMove(m), depth − 1, opponent(onTurn),
   −beta, −alpha)
8:         if value > bestValue then
9:             bestValue ← value
10:        end if
11:        if value > alpha then
12:            alpha ← value
13:        end if
14:        if alpha ≥ beta then                                ▷ Pruning (cutoff)
15:            break
16:        end if
17:    end for
18:    return bestValue
19: end function
```

2.1.3 Random Moves via Alpha-beta

When a robot plays a strategic game without randomness many times, it can be observed that the robot moves are deterministic, i.e., in the similar situations it plays a similar move, because the search tree is similar. Thus the robot opponent can build up a strategy how to play against it by playing many times with the robot (particularly when the starting position is still the same). Hence a randomized playing is needed to be done for the robot.

On the other hand, randomization of moves should not lead to silly moves and thus a search is needed. So we want to choose a move with a value not far from the value of the best move, i.e., with a value at least $bestMoveValue - margin$ for a given constant $margin$. We call such moves *good enough moves*. We propose an algorithm for searching for good enough moves using Alpha-beta.

The search only differs from Alpha-beta in the root node, in other nodes there still runs deterministic Alpha-beta. In the root node we add to the search function collecting all moves together with their value into a list. After searching all moves, the list is filtered to have only good enough moves, i.e., with a value in range from $bestMoveValue - margin$. Between these nodes the random is chosen uniformly, e.g., using a random number generator.

There is one modification of the recursive calling of Alpha-beta too (for clarity Alpha-beta is now not in the Negamax form). Suppose that the best move is searched first and there are some other good enough moves. While searching these moves in the second level of the tree from the root, *alpha* is $bestMoveValue$ (best value that Max can achieve) and *beta* is updated. For example it can happen that *beta* changes to $bestMoveValue - 5$ and the search in this node is pruned, so move is considered to be good enough), but the real value of the move is

$bestMoveValue - 100$ and the move is in fact not good enough. To avoid these false positives *alpha* should be first set to $bestValue - margin - 1$.

This leads to Algorithm 3 executed only for the root node (parameters *alpha* and *beta* are useless in the root node, but one can add a break in the case when wining move is found).

Algorithm 3 Random Alpha-beta algorithm

```
1: function alphaBetaRandom(position, depth, onTurn)
2:     bestValue ← −∞ − 1
3:     for all m in generateMoves(position, onTurn) do:        ▷ Try all moves
4:         value ← alphaBeta(executeMove(m), depth − 1, opponent(onTurn),
   −∞, −bestValue + margin + 1)
5:         if value > bestValue then
6:             bestValue ← value
7:         end if
8:     end for
9:     return bestValue
10: end function
```

Up to my knowledge nobody has done choosing random moves using Alpha-beta in this way, so this algorithm for choosing random moves via the Alpha-beta search is a contribution of this thesis.

2.1.4 Iterative Deepening

Iterative deepening (ID) is simply running the Alpha-beta algorithm first to depth 1, then to depth 2, 3, ... Searching one ply deeper lasts in most games much more than the previous search, so the deepest search takes nearly the whole time of all searches, hence ID does not slow down the search. It is used to estimate how deep a robot can search within a time limit. The implementation of that estimation for Tzaar is briefly described in Section 3.4.

2.1.5 Transposition Table

Transposition table (TT) is in fact a hash table for storing information about positions. In many games it happens quite often that two positions can be reached by two different move sequences. In these cases storing some information about searched positions comes in handy.

Nevertheless, there are some differences between the Transposition Table and a usual hash table. Since the whole position representation is large, we cannot store the position in TT. Instead of it we generate a big enough hash value for the position (64 bits are mostly sufficient) and only a part of the hash value is used as an *index* in the table, i.e., the position in the array in which TT is stored. Thus it is handy to have the size of TT 2^k for a constant k.

Since the number of searched positions is very high (up to billions), there can occur two types of collisions: two different positions obtain the same hash value — this is called *type-1 error* — or two positions obtain the same index in TT — *type-2 error*. Type-1 error is a lot more critical than type-2 error, so the

hash value range should be very high and the hash function has to uniformly and randomly give values to the positions. This error is often rare and it is tested only by trying to play saved moves from TT in the position. It is usually not tested in another way, since the test would consume too much memory and would slow down the search.

Type-2 error occur quite often, because the size of TT is, limited by the memory and usually does not exceed millions. This error can be found by comparing the whole hash value stored in TT

The problem with type-2 error occurs when we want to save a position into TT and there is another position saved on the index. If we just delete the position that was on the index, we may rewrite a search result that was counted for a long time by a result calculated quickly. Thus we use a *replacement scheme Two big* (for more replacements schemes and comparison of them in the domain of Chess see [3]). In the scheme Two big we count how many positions were searched to obtain a value of a position and store it to TT. On each index there are two positions in the table. Newly inserted position to TT is always stored and when there are already two positions under one index, we delete the one with fewer nodes searched to obtain its value.

In whole we want to save to TT for each position the hash value, the counted value, the best move, the search depth and the number of nodes examined when searching the position. The counted value can be one of three types: *lower bound* when the value is lower than *alpha*, *upper bound* when the value is bigger than *beta* (the search was pruned in the position), or *exact value* otherwise. The type of the value is also saved.

When we want to search a position in depth $d > 0$, we first look into TT whether it was searched to the sufficient depth. If so and the type of the value is exact value, the position need not be searched. Otherwise it is searched and we also update the bound *alpha* or *beta* when the type of the value is lower bound, or upper bound, respectively. After the end of the search we save a result of it, i.e., a value, the type of the value and the best move, together with some information about the search into TT.

The only theoretical thing left around TT is the hash function. For board games *Zobrist hashing* [21] is usually used. For each possible combination of figure and field on the board (including empty fields), a random value in the range of hash values is generated. For example for Tzaar a random value is generated for each combination of a stack type, color, height and a field. The hash value of a position is xor of values of all combinations of a figure and a field on the board. The advantage of the Zobrist hashing is that it can be counted incrementally, i.e., after a move the value is changed by xor with what has disappeared from the board and what has appeared.

2.1.6 Move Ordering

The crucial thing about Alpha-beta algorithm is to have moves in the order from the best to the worst. Since we do not know which move is the best, we have to try some heuristics.

The first and probably the most important heuristic comes from using the Iterative Deepening and the Transposition Table. Suppose we are not in the first

iteration of ID, then we have already searched internal nodes of the search tree and stored for each of them the best move in the previous iteration of ID. The best move can change in the next iteration, since we search the position deeper, but we can use the stored moved as the first. This heuristic is often called the *Hash Move* (the move is stored in TT) or the *Principal Variation Move* (it was the best in the previous iteration of ID).

If the search is not pruned by searching the Hash Move (if it exists), we have to try another moves and we want to do it in a good order. One way how to do it is generating moves already in a good order, the other way is first generate all moves, assign them heuristic value and sort them according to the value.

2.1.7 History Heuristic

The *History Heuristic* is a dynamic method for making the Move Ordering better. It is based on counting how many times each move caused a cutoff (the search was interrupted in the current node), e.g., by using quadratic array with one coordinate as the source field of the move and the other coordinate as the destination field. We can now observe that moves which caused a cutoff many times should be tried before the other moves. One possible implementation is to add the number of cutoffs to the heuristic value for sorting generated moves.

Another important heuristic in many Chess programs, called the *Killer Move*, is trying a few moves that caused a cutoff in another nodes in the same depth of the search tree. It is similar to the History Heuristic.

2.1.8 NegaScout

The *NegaScout algorithm* comes with a following idea. We restrict ourselves and try to search a move. If we still obtain a value at most *alpha*, or at least *beta* (thus a cutoff in the second case), we do not need to search the move again without the restriction. Otherwise the re-search of the move is needed.

The restriction is that we set *beta* temporarily to $alpha + 1$. Since the range between *alpha* and *beta* is called *window*, this restriction is called *null window* (the window is as narrow as possible). We can observe that the search with the null window is often much quicker that with the full window, i.e., without restrictions. The whole NegaScout algorithm is more effective than Alpha-beta (effectiveness means number of nodes examined), but relies on a good Move Ordering. The proof of its correctness can be found in [14].

Algorithm 4 shows the pseudocode of the NegaScout.

2.1.9 Quiescence Search

Alpha-beta stands on a good leaf evaluation. Since the evaluation should be quick, it can often happen that a leaf obtain a high value, but the value change dramatically with the next move (which is not examined). This problem is called *horizon effect*.

To resolve the problem, Quiescence Search is used. In every leaf a restricted search is run and the restriction is that we use only moves which can change value dramatically (like capturing high stack in Tzaar). The leaves of the search

Algorithm 4 NegaScout algorithm

```
1: function NEGASCOUT(position, depth, onTurn, alpha, beta)
2:     if depth = 0 or endOfGame(position) then                    ▷ leaf node
3:         return evaluate(position)
4:     end if
5:     bestValue ← −∞ − 1
6:     beta2 ← β                                  ▷ Full window for the first move
7:     for all m in generateMoves(position, onTurn) do:            ▷ Try all moves
8:         value ← negaScout(executeMove(m), depth − 1, opponent(onTurn),
   −beta2, −alpha)                                                 ▷ Null window search
9:         if value > alpha and value < beta and m is not the first move then
10:            value ← negaScout(executeMove(m), depth − 1,
   opponent(onTurn), −beta, −alpha)                                ▷ Full window re-search
11:        end if
12:        if value > bestValue then
13:            bestValue ← value
14:        end if
15:        if value > alpha then
16:            alpha ← value
17:        end if
18:        if alpha ≥ beta then                                    ▷ Pruning (cutoff)
19:            break
20:        end if
21:        beta2 ← α + 1                                           ▷ Update beta2
22:    end for
23:    return bestValue
24: end function
```

tree are *quiet*, i.e., there is no move that changes the value much. For more information see e.g. [17].

2.2 Proof-number Search

Proof-number Search (PNS) is a best first search algorithm for finding the winning strategy in game trees developed by Allis [1]. In a sufficient time and memory it can decide whether a given position is winning for us, or for our opponent (suppose we are on turn). Ties can be viewed as a loss for us. For simplicity and their non-existence in Tzaar we do not mention them.

During the search we have a part of the game tree stored in the memory, we call this part *expanded tree*. A node in the tree can have three values: *true* if it is won for us, *false* if it is won for the opponent and *unknown* otherwise. The tree is viewed as an AND/OR tree, i.e., an internal node is OR when we are on turn, or AND when our opponent is on turn and the root node is an OR node. OR means that we can choose any move leading to a win, but in opponent's AND nodes we must examine all his moves to be sure the node is won for us. Note that the algorithm in general works also on arbitrary AND/OR trees [1], but we deal only with game trees.

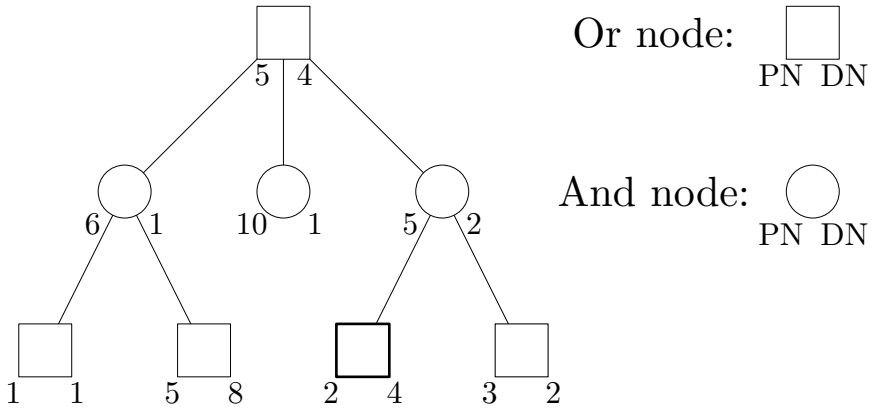


Figure 2.2: Example of a part of an AND/OR tree with proof and disproof numbers. Circles denote AND nodes and squares OR nodes. The highlighted OR node is the most proving node in this tree.

We want to find a proof in the tree that corresponds to a winning strategy for us, or disproof when there is no winning strategy. The tree can also be viewed as a very huge formula for which we want to decide whether it is true or not.

The main idea of the algorithm is to look for the shortest proof and disproof. We store a *proof number* and a *disproof number* for each node in the expanded tree. The proof number (PN) of a node N is the minimum number of nodes that we have to prove to obtain the value true in the node N. The disproof number (DN) is similarly the number of nodes we have to disproof to have the value false in the node N.

In leaves of the tree, nodes obtain proof and disproof numbers straightforwardly: PN is 0 and DN is ∞ for a node with value true (no node has to be proved to prove the node and the node cannot be disproved), PN is ∞ and DN is 0 for a false node and both PN and DN are one for an unknown node.

For an internal OR node, PN is minimum of PNs of its children, since we can choose where to move to win, and DN is the sum of its children. When we sum DNs and there is ∞ between the children, the resulting DN is also ∞. For an internal AND node the computation is done vice versa: PN is the sum of children's PNs and DN is the minimum of children's DNs. We can see that an OR node has the value true if and only if PN is 0 and DN is ∞ and vice versa holds for an AND node. For an example see Figure 2.2.

The PNS algorithm can be basically described by a loop done until a root node has the value unknown. In the loop, three steps are executed:

1. find an unknown leaf node and expand it, i.e., add all its children to the expanded tree,

2. assign to the children values unknown, or true, or false,

3. update proof and disproof numbers in the tree.

The second step is obvious to implement and the third step is done by updating PNs and DNs on the path from the expanded node to the root node (in this order) as PNs and DNs of the other nodes were not changed.

The choice of a node to expand is done according to the searching for the shortest proof and disproof. We want to find a leaf node for which it holds that

proving it decrements PN and disproving it decrements DN of the root node. Such a node is called *the most proving node* (MPN). We select it by walking on a path from the root until we get to a leaf and find MPN. In OR nodes we continue to the child with the lowest PN, in AND nodes we go to the child with the lowest DN. For an explanation of such a selection see [1].

Note that the algorithm runs the best on trees which are not uniform, for example when one player repeatedly threats the other player who has only a few possible moves that do not lead to a quick loss. So the algorithm often finds a winning strategy that is in fact a long sequence of threats and the length of the strategy in moves may be far bigger than the length of the shortest winning strategy.

The algorithm can be simplified in a way similar to the Negamax algorithm. Let all nodes be OR and the only thing we have to do is swapping PN and DN between tree levels. See a pseudocode of the Depth-first Proof-number Search in Algorithm 5.

Main disadvantage of this algorithm is that it has the whole search tree in the memory. Many improvements that have lower memory requirements were invented, see [6] for a survey. We describe an adjustment of PNS that is up to my knowledge the most used nowadays.

2.2.1 Depth-first Proof-number Search

Depth-first Proof-number Search (DFPNS) is PNS adjusted to the depth-first search and can be implemented using one recursive function. In the memory we store only the path from the root to the current MPN, the other nodes already searched are in the Transposition Table (TT) similar to the one used in Alpha-beta. Nodes in TT can be also deleted in the case of a collision.

We also postpone the update of PN and DN of nodes upper in the search tree while MPN is still in their subtree. We do it by using thresholds on PN and DN. When the search selects a child, it sets the thresholds such that when PN or DN is at least the corresponding threshold, MPN is not in the child's subtree.

We show a formula for counting the thresholds for a child of an OR node. Let tpn and tdn be thresholds on PN and DN of the current OR node, $dn1$ be DN of the child with the lowest PN between children, $pn2$ be the second lowest PN and $sumDN$ be the sum of all children's DNs. Then the new threshold on PN is the minimum of tpn and $pn2 + 1$ and the threshold on DN is $tdn - sumDN + dn1$.

Note that the child with the minimum PN has MPN in its subtree (or it is already MPN), so the search goes to it. For an AND node the computation is done vice versa. The thresholds for the root node are set to ∞.

The Transposition Table plays important role in this algorithm. When we count non-trivially PN and DN of a node and MPN is not in the subtree of the node, we store PN and DN for that node to TT. When we are searching between children of a node (to count minimum of PNs and the sum of DNs), we look for child's PN and DN first to TT, if the child is not a final position. If PN and DN for the node are not in TT, we set PN and DN to 1.

Algorithm 5 shows the pseudocode for DFPNS adjusted similarly to Negamax.

Algorithm 5 DFPNS algorithm

```
1: procedure DFPNS(node, onTurn, tpn, tdn)
2:     minPN ← ∞                                          ▷ Minimum of PNs
3:     dn1 ← 0                                  ▷ DN of child with minimum of PNs
4:     minPNnode ← null                              ▷ Node with minimum of PNs
5:     pn2 ← ∞                                          ▷ Second minimal PN
6:     sumDN ← 0                                                ▷ Sum of DNs
7:     loop until break
8:         for all m in generateMoves(node, onTurn) do          ▷ Try all moves
9:             child ← executeMove(m)
10:            pn ← 1                          ▷ Default values for unknown leaves
11:            dn ← 1
12:            if endOfGame(node) then
13:                if winner == onTurn then                              ▷ Our win
14:                    addNodeToTT(node, 0, ∞)          ▷ PN is 0 and DN is ∞
15:                    return                  ▷ No need to search anything else
16:                else                                                  ▷ We lose
17:                    pn ← ∞
18:                    dn ← 0
19:                end if
20:            else
21:                ttEntry ← lookupInTT(child)
22:                if ttEntry ≠ null then                ▷ Already searched node
23:                    pn ← ttEntry.dn                ▷ Note swapping PN and DN
24:                    dn ← ttEntry.pn
25:                end if
26:            end if
27:            if sumDN < ∞ then
28:                sumDN ← sumDN + dn
29:            end if
30:            if pn < minPN then
31:                pn2 ← minPN
32:                minPN ← pn
33:                dn1 ← dn
34:            else if pn < pn2 then
35:                pn2 ← pn
36:            end if
37:        end for
38:        if minPN ≥ tpn or sumDN ≥ tdn then                ▷ Test thresholds
39:            addNodeToTT(node, minPN, sumDN)
40:            break
41:        end if
42:        ntpn ← tdn − sumDN + dn1                ▷ Again swapping PN and DN
43:        ntdn ← min(tpn, pn2 + 1)
44:        dfpns(child, opponent(onTurn), ntpn, ntdn)
45:    end loop
46: end procedure
```

Note that there are some differences to the pseudocode of DFPNS in [5], e.g., we do not need to call function `dfpns` for final positions and we can cut off the search immediately when we find a win.

We can also observe that DFPNS searches the tree in the same order as PNS when TT is big enough and no node is deleted from it.

There are some properties of DFPNS that can be adjusted to get more effective algorithm. The first is the initialization of unknown leaves to more appropriate numbers than one, the second is better setting the thresholds for the selected child and the third is the way of counting disproof numbers in an OR node.

Next we describe most of DFPNS enhancements that were invented – the others are not so important for Tzaar, since they are dealing with problems not occurring in Tzaar. We show computations only for OR nodes, because the pseudocode in Algorithm 5 is modified in the Negamax way, i.e., every node counts PN and DN like an OR node, and changing formulas for AND nodes is straightforward.

We note that there is another problem in some games called *Graph History Interaction*. It is when the current state of a position depends on a few or even all positions that were in the game recently. The problem makes use of the Transposition Table harder. Since this is not the case of Tzaar, we do not describe any method how to deal with it. For a solution see e.g. [9].

2.2.2 Evaluation Function Based PNS

Evaluation Function Based PNS is an enhancement based on the better initialization of unknown leaves. It was originally proposed by Schadd and Winands [15] for PNS, but it can be easily added to DFPNS.

Since mostly more than one node has to be searched to determine the value, we want to take into account the branching factor (at least an estimate of it) and also who has an advantage – that is estimated by the evaluation function. The evaluation function can be relatively slow, so we count the value for the searched node instead of for each of its children. Then we use a step function

$$step(value) = \begin{cases} 1 & \text{if } value \geq t \\ 0 & \text{if } -t < value < t \\ -1 & \text{if } value \leq -t \end{cases}$$

where t is a threshold parameter corresponding to a value that indicates player's high advantage, i.e., the player is likely going to win. Using this function we initialize the proof and disproof numbers for an AND leaf node with this formulas:

$$pn = m \cdot (1 + b \cdot (1 - step(value))),$$

$$dn = 1 + a \ \cdot (1 + step(value))$$

where m is number of moves from the child (or an estimate of it) and a, b are parameters. Initialization for an OR leaf is done vice versa.

2.2.3 $1 + \epsilon$ Trick

The problem which leads to using the *1 + ϵ Trick* is switching between subtrees where the most proving node (MPN) is. The worst case is when two children of an OR node have nearly the same proof numbers and their subtrees cannot be stored in Transposition Table together. Then it can happen that proof numbers of these children increase repeatedly only by one, making the algorithm to switch between their subtrees very often and recalculate PNs and DNs of some nodes many times. The solution of this problem was proposed by Lew and Pawlewicz [12].

To decrease the number of switches between subtrees we change the setting of the threshold on PN for a child of an OR node to the minimum of *tpn* and $pn2 \cdot (1 + \epsilon) + 1$ for a constant $\epsilon > 0$. The same formula with disproof numbers instead of proof numbers is used in AND nodes.

2.2.4 Weak PNS

The search tree is often not in fact a tree, but a directed acyclic graph because of positions that occurs more than once in the tree. Then DFPNS suffers from *double-counting problem*, i.e., the problem when the proof number of a position contains the proof number of another position twice or even much more times.

The problem can be resolved by changing the summing of disproof numbers in OR nodes and proof numbers in AND nodes. *Weak PNS* [5] proposes taking the maximum disproof number and adding the number of children minus one. Another solution to this problem is described by Kishimoto [9].

2.2.5 Heuristic Weak PNS

Next we propose an enhancement based on Weak PNS and the evaluation function which can be viewed as a contribution of this thesis. We modify counting disproof numbers in OR nodes in a way similar to Evaluation Function Based PNS. The idea of using evaluation function is briefly mentioned by Kishimoto [9] when comparing Weak PNS to algorithms described there, but Kishimoto probably did not use a step function.

The definition of a step function is similar to the one in Evaluation Function Based PNS.

$$step(value) = \begin{cases} 2 & \text{if } value \geq t \\ 1 & \text{if } -t < value < t \\ 0 & \text{if } value \leq -t \end{cases}$$

where t is a parameter corresponding to a value that indicates a player's high advantage, i.e., the player is probably going to win. We compute the disproof number with this formula:

$$dn = maxDN + step(value) \cdot h \cdot (m - 1)$$

where $maxDN$ is the maximum DN between children, m is the number of moves and h is a positive constant.

Now we discuss why this formula is better than the one in Weak PNS. When a player on turn has a big advantage and *value* is at least t, we probably have to

search many nodes to disproof the node. Since the player is likely going to win, the disproof number is probably ∞. Thus we can set DN to $maxDN+2\cdot h\cdot(n-1)$. In the case of balanced position, i.e., no player has a significant advantage, we count DN nearly the same as in Weak PNS, only with factor h. Because of this, the parameter h should not be much higher or lower than one. When a player on turn is in a bad position and *value* is at most $-t$, we likely do not need to search many positions to disproof the node.

2.2.6 Dynamic Widening

Technique similar to Weak PNS that also tries to avoid the double-counting problem and overestimation of DN or PN is the *Dynamic Widening* invented by Yoshizoe [20]. In an OR node instead of summing all children's DN we sum only DN of the J best children. The best means that the J children are the first J children in the increasing order by PN. This leads to the formula:

$$dn = \sum_{i=1}^{J} dn_i$$

where dn_i is a disproof number and pn_i is a proof number of i-th child and $pn_1 \leq pn_2 \leq \ \cdots \ \leq pn_n$. The parameter J can be a constant or it can depend on the number of children.

2.3 Alpha-beta and DFPNS in Lost Positions

Suppose we are not in the final position and Alpha-beta has found out that the position is lost, or DFPNS has found disproof. Since our opponent can overlook his win, we want to make a move that is still good despite the lost position. Since we know nothing of our opponent, the playing heuristic in lost positions should be general. I have not found anything about it in literature, so it is a contribution of this thesis.

One possible solution is to make as much chaos as possible, but this is very domain dependent heuristic and I was not able to make up how to do it in Tzaar.

More general solution is to find a move that leads to loss after the maximal possible number of moves. By doing that it is more probable that the opponent makes a mistake. When we use the Iterative Deepening in Alpha-beta, we just take the best move in the last iteration in which Alpha-beta has not found out that the position is lost.

Looking for the deepest loss in DFPNS is also possible, but we cannot guarantee that it finds the real deepest loss, since it does not need to search every position in the depth of the deepest lost position and higher in the search tree. Anyway we can add counting of the maximal losing depth and the minimal winning depth which is pretty similar to the counting of proof and disproof numbers.

A leaf won for us obtains the maximal losing depth ∞ and the minimal winning depth 0 and vice versa for a leaf lost for us. For an unknown leaf the maximal losing depth is 3 (it is the worst case) and the minimal winning depth is ∞, since we do not know anything. An internal OR node has the minimal winning depth

the minimum from maximal losing depths of his children plus one, and the maximal losing depth the maximum of the minimal winning depths of his children plus one.

With this computation we can find a lower bound on the maximal losing depth of the root node and a move corresponding to it which can be used in the case when DFPNS finds disproof.

3. Algorithms on the Domain of Tzaar

In Section 1.4 we discussed properties of the game Tzaar, mainly the large branching factor due to two moves in a turn of each player and short games which is typically up to 28 plies. According to these properties we discuss which algorithms are suitable for playing Tzaar. Their implementation and domain dependent heuristics follows.

3.1 Implementation of Tzaar Board

First of all we show an implementation basis of our robot. The Tzaar board is a hexagon with five fields on each side, but we cannot simply store it in the memory as a hexagon. Thus it is sloped to be fit in the quadratic array 9×9.

Example of the array containing the fixed starting position (array members are separated by spaces):

```
-1   1   1   1   1 100 100 100 100
-1  -2   2   2   2  -1 100 100 100
-1  -2  -3   3   3  -2  -1 100 100
-1  -2  -3  -1   1  -3  -2  -1 100
 1   2   3   1 100  -1  -3  -2  -1
100  1   2   3  -1   1   3   2   1
100 100  1   2  -3  -3   3   2   1
100 100 100  1  -2  -2  -2   2   1
100 100 100 100 -1  -1  -1  -1   1
```

The number 100 stands for a field outside the board and also for the field in the center of the board. An empty field with no stone has number 0. Other numbers stand for different types of stones: 1 is a white Tott, 2 is a white Tzarra, 3 is a white Tzaar and black pieces have the same numbers multiplied by -1.

In the array board a player can move in six directions: horizontally left, or right, vertically up, or down, and diagonally right and down, or left and up. The other diagonal directions (right and up, left and down) are not possible.

Heights of stacks are in another array of the same size and format. In these arrays the field usually denoted by A1 is in the left upper corner (on the index 0) and field I5 on the index 80.

Note that during the search there is only one position stored in the memory. The program remembers also some additional information of the position. The evaluation function and the other algorithms use some positional and material information, e.g., the hash value, the highest stacks of each type and the zone of control. This can be counted statically, i.e., for each position anew, but it would slow down the search very much, thus it is counted incrementally, i.e., the value is quickly changed when a move is executed.

The zone of control determines how many stones of a certain type can be captured by one move, no matter who is on turn. It is used by the evaluation function and for determining whether a player on turn has lost because of no

possible captures – the zone of control of all opponent's stone types is zero in this case.

Details about implementation and the program can be found in the documentation, see Appendix A. The implementation of the algorithms is described in Section 3.3.

3.2 Algorithms for Tzaar

Since the game tree properties differ in the middlegame and in the endgame, we discuss these parts of the game separately. Also we choose between knowledge based methods and brute force methods.

Some Chess computer programs use an Opening Book [34], i.e., tables with many different openings, to play quicker and better in the beginning of the game. In Tzaar a lot of games start with a random starting position and the number of starting positions is very large, exactly $5.94 \cdot 10^{39}$ (see Section 1.4). Hence building the Opening Book is not possible. Even if we consider only one starting position, e.g., the fixed starting position, there are many reasonable moves, it could be one fourth of all moves, thus the Opening Book for only the first six plies would be very large. The opponent also can make program fall out of the book doing a weaker move and this weaker move would not result in a position much better for our robot if it is done early in the game. Instead of the Opening Book we use random Alpha-beta proposed in Section 2.1.3.

The Endgame Tablebase [33] is not reasonable too, since the number of positions with only six pieces is $6.22 \cdot 10^{14}$ which is too much. These positions can also be quickly solved by a simple Minimax search and the number of positions with more than six pieces is even larger. The Opening Book and the Endgame Tablebase are knowledge based method. According to the presented arguments together with the state space and game tree complexities and conclusions of Heule and Rothkrantz [7], we can conclude that knowledge based methods are not suitable in the domain of Tzaar and our program have to use a brute force search.

Opening has the largest branching factor, but it is not very different from the middlegame. Before the endgame a player (attacker) mostly cannot capture defender's high stack or even win in a few moves by a threat sequence. From the observation in Section 1.3 defender can escape with his stack from most of threats easily and there are often more different possibilities how to do it. Thus we can conclude that algorithms based on threats would be ineffective during the opening and middlegame, thus Proof-number Search is not used before the endgame.

The only algorithm left that is known to me is Minimax with the Alpha-beta pruning and some of its enhancements. The Iterative Deepening is implemented because of time estimation, i.e., how deep can the program search, and because of the Move Ordering. The Transposition Table is used for storing moves from the previous shallower search (the Principal Variation Move) and also because some positions can be reached by a few different move sequences.

The domain specific Move Ordering and the History Heuristic were implemented too. To find cutoff nodes quicker, NegaScout is used. See Section 4.1.1 for information how each enhancement makes the search quicker.

From important enhancements of the Alpha-beta algorithm the Quiescence Search was not implemented. The main reasons are that in most positions there is no move which changes the value much and that searching for it would probably mean to generate all moves which would cost a lot of time. Instead of that, the concept of the zone of control is used in the evaluation function. Killer Move heuristic was also not implemented.

In the endgame the branching factor is not so high and threat sequences can occur more frequently. There are also not so many solutions to threats, thus threats limit the branching factor and the Proof-number Search (PNS) can be sometimes more effective than the Alpha-beta search.

PNS as proposed by Allis [1] is very memory consuming and search trees in Tzaar endgames are too large to fit in the memory. For these reasons the Depth-first Proof-number Search (DFPNS) is used.

Like the Alpha-beta algorithm, DFPNS has also some enhancements and each of them is important. The first is the heuristic initialization of leaves based on the evaluation function. The $1 + \epsilon$ Trick is also implemented to avoid the situation when the search jump across the tree very often. DFPNS sometimes suffer from the overestimation of proof and disproof numbers, thus Weak PNS and the Dynamic Widening were implemented. Weak PNS is also modified to Heuristic Weak PNS, but from the results in Section 4.1.4 we can see that the Heuristic Weak PNS does not have better search duration or solvability than the Weak PNS or the Dynamic Widening.

There are some other algorithms searching for the best move in board games. For the Lambda Search [18], I was not able to think up how to determine quickly the order of a threat, thus it is not implemented. Since there is mostly more than one possibility how to escape from a threat, we can conclude that the Dependency-based Search [1] is not suitable for Tzaar. The Monte Carlo Tree Search [2, 11] is probably worth trying for Tzaar, but it is not implemented at all.

3.3 Implementation of the Algorithms

In comparison with Chess the main difference in Tzaar is that there are two moves in each turn of a player (except for the first turn of white player). For simplicity moves are generated separately for the first and the second move of a turn and they are executed and reverted also separately. In Alpha-beta and DFPNS functions there are two nested loops, one for the first moves and the other for the second moves that are generated after doing the first move.

Also the depth in the search tree in Alpha-beta and DFPNS is counted in moves, not turns of a player. The program has often time to search the tree with the Alpha-beta algorithm to the depth 5 (2 and half plies), but not to the depth 6 (3 plies) because of the large branching factor. Hence the Iterative Deepening increases search depth by a half ply, i.e., one move. In DFPNS a maximal depth in which a player lose (see Section 2.3) is counted similarly – the depth is the number of moves.

Again for simplicity, searching functions assume that the first turn of white player consists of two moves as the other turns. This is not in fact true, but it does not result in worse play and does not slow down the search much.

The Transposition Table (TT) differ for Alpha-beta and DFPNS. TT for both algorithms stores for each position the hash value and the number of nodes searched to obtain a value of a position (visited nodes in the subtree of the position). Alpha-beta saves the best moves, the counted value with the value type (exact value, lower bound or upper bound) and the depth to which the position was searched. DFPNS saves the proof and disproof numbers, the minimal winning depth and the maximal losing depth.

The Alpha-beta pruning enhancements are implemented straightforwardly with the exception of the History Heuristic. The History Heuristic is counted only for the first move of a turn, because the array with both moves would have four dimensions (from and to field for the first and the second move) and thus would be very large – precisely $81^4 = 43\,046\,721$.

Note that the Principal Variation Move stored in TT from the previous iteration of ID is tried before generating moves and if it leads the cutoff, we do not need to generate moves at all.

Enhancements for DFPNS are implemented straightforwardly too. DFPNS uses the heuristic Move Ordering described in Section 3.6 which, quite surprisingly, helps to solve some positions, see Section 4.1.3.

The tournament version of Tzaar is not implemented, because it is also not implemented on servers where Tzaar is played and I do not know which strategy the robot should follow in the placement phase.

3.4 Search Time Estimation

The program has limited time to search and the limit depends on robot's level. For Alpha-beta the Iterative Deepening (ID) is used to search to the maximal depth that can fit into the time limit. To estimate the duration of searching to the next depth the duration of the last iteration of ID is multiplied by an expected branching factor.

The expected branching factor is a little less than the real branching factor because of pruning and depends on the last move done in leaves of the search tree, i.e., whether it was the first or the second move of a player, and on the number of stacks on the board. Figure 3.1 shows the multipliers for both moves of a turn.

DFPNS is usually run until it finds a proof or a disproof. To fit the search into the time limit it is stopped after searching a certain number of nodes. To estimate how many nodes can DFPNS search in a given time, at first a certain number of nodes, e.g., 100 000, is inspected and then according to the duration of searching them the maximal number of nodes to search within the given time is counted.

3.5 Evaluation Function

The evaluation of a position in Tzaar is used both by the Alpha-beta search and DFPNS. It consists of a material value for each player and a positional evaluation. The value is $+\infty$ when the position is won by white player and $-\infty$ if it is won by black player. Balanced positions, i.e., positions where no player

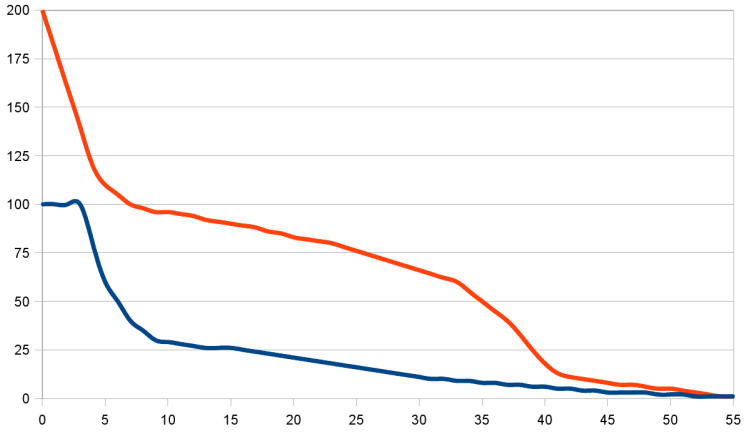


Figure 3.1: Graph of multipliers for ID for the first move (blue) and the second move (red) according to the number of free fields.

has an advantage, obtain number near to zero. In positions with value greater than zero, white player has an advantage or even is going to win and vice versa, values less than zero means an advantage for black player.

The material value together with some positional information is counted incrementally, other positional features are counted statically for each leaf node that is not won by a player.

The material value is the sum of values of all stacks on the board. Values of white stacks are positive, black stacks have negative values counted by the same function as the white stacks. A stack obtain a value according to following formula: $StackHeightValue$[stack height] $\cdot$ $StackByCountValue$[count of stones of the same type], where $StackHeightValue$ determines the value of a stack according to its height. Values of this array are shown in Figure 3.2.

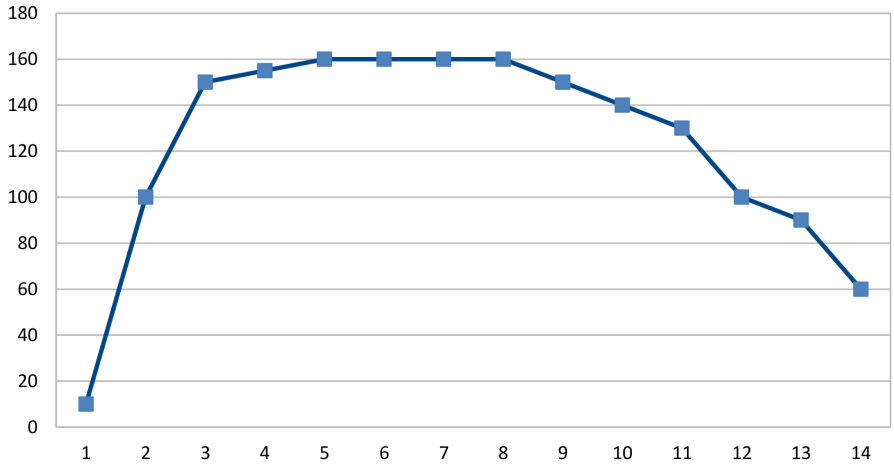


Figure 3.2: The value of a stack according to its height.

The array $StackByCountValue$ contains values of a stack according to the number of other stacks of the same type, see Figure 3.3.

Some other formulas and constants were tested, but their use result in worse

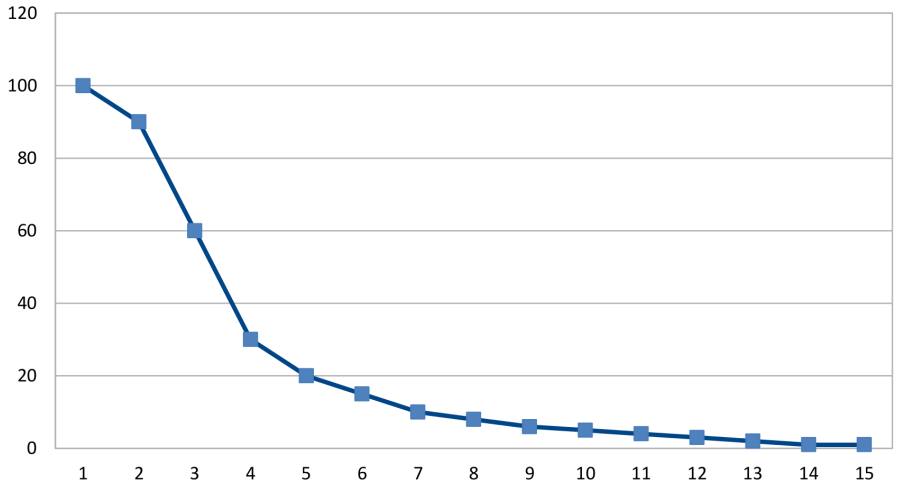


Figure 3.3: The value of a stack according to the count of pieces of the same type.

play.

Since stacks with the height greater than one are more valuable in the middle of the board, they obtain a little bonus (up to 25) when they are not on the margin of the board. The worst position for a high stack is a corner, thus a high stack in the corner decreases value by 30. This is the only part of the positional value which is counted incrementally.

The other part of the positional value is counted statically, i.e., for each node of the search tree again, but the static value function uses information which are maintained incrementally, e.g., the highest stacks for each type and the zone of control. Probably the most important part of the positional value are threats – when a player has only at most two stacks of one type and they are both in his opponent's zone of control (can be captured with one move), the opponent obtain a bonus of at least 1 000. If the opponent is on turn, the bonus is 2 000 000, because the opponent is in a winning position.

Having bigger zone of control is a little advantage, thus a player gets some points for threatening opponent's stacks according to this formula: stacks in zone of control · (initial stone count - count) / initial stone count.

When a player can capture only few stacks he is likely going to lose because of no possible captures, so his opponent obtain a value of 100 000.

There are two static value features concerning on stack heights. A player get 25 000 for each stone type that is "secure" – that means for which a player has a stack higher than all stacks of his opponent. If all player's stone types are secure, a player obtain 500 000. Also having a stack with height two or more of all types can be an advantage, thus a player get 100 000 for this.

We remark that the most of the evaluation function was done by inventing features and setting constants intuitively according to the observations done in Section 1.3 and then playing with the robot. Different versions of the robot were also tested in games against each other.

3.6 Move Ordering

The Move Ordering is done straightforwardly by generating moves into an array, assigning them a heuristic value and simply sorting them by the value using QuickSort algorithm. The less the sort value is, the better move should be and the sooner is executed. Moves for a player on turn are generated separately for the first and the second move of a turn and assigning a value to the moves also differ for the first and the second move.

The first move is always a capture. Capturing a stack of a type that occurs on the board quite often, i.e., there are relatively many visible stones of that type, is not very advantageous, thus the sort value is linearly dependent on the count of stacks of the captured type (the multiplier is constant $SortCaptureCountMult$ with default value 10).

The height of the captured stack is also taken into account and it is more significant than the count of stacks of the stack type. The dependence on the height grows very fast with the height less than six (value is up to 200) and then it is approximately linear, see Figure 3.4. The value for the height is subtracted from the value for the count.

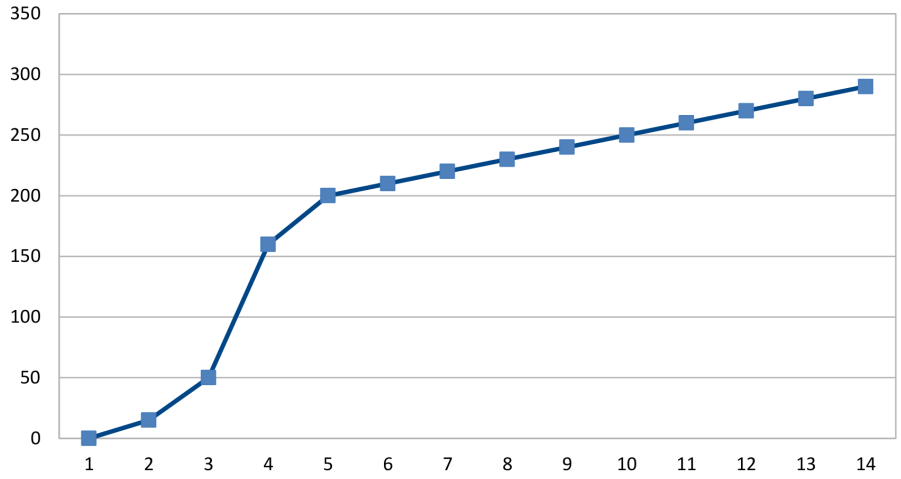


Figure 3.4: The value of a stack according to its height in the Move Ordering.

For the first move the History Heuristic is also taken into account. The number of prunes caused by a move, precisely by the move with the same fields from and to, is multiplied by the constant $SortHistoryPruneMult$ (default value is 50) and subtracted from the value.

The second move can be a capture too, but often it is a stacking move and rarely a pass move. From the observation in Section 1.3 stacking is mostly the best choice and stacking moves should be done before captures of stacks with the height one. Capturing stacks with height at least three is mostly better than doing a stack move. Passing is reasonable only in the endgame and this move should be tried at last.

For a capturing move the sort value is counted in the same way as for the first move. For a stacking move the value is linearly dependent on the count of stacks of the same type as the resulting stack has and the multiplier is the constant $SortStackCountMult$ (default value is only 3). Since capturing stack types with

small count on the board can often lead to a win, the constant *SortStackBonus* (with default value 19) is added to the value of a stacking move.

Different values of constants are tested in Section 4.1.2.

3.7 Robot Levels

As shown in Chapter 4, the implementation of these algorithms leads to a robot that is able to play on the level of best players on Boiteajeux.net. Thus for not advanced players we want to have robots that are not so strong and we create four levels: beginner, intermediate, expert and unbeatable. We describe in what circumstances the robots use which algorithms.

The expert and unbeatable levels have all features mentioned above (Alpha-beta to the depth according to the time estimation, DFPNS, the best evaluation function ...), they only differ in the time limit, the expert level has 30 seconds (this is sometimes a little exceeded) and the unbeatable level has 300 seconds. For the first three plies the random Alpha-beta proposed in Section 2.1.3 (without the History Heuristic and the NegaScout) is used with margin 20, then we run Alpha-beta without randomness, but with all enhancements. When the number of stacks is at most 23, DFPNS is used. If it does not find a solution, Alpha-beta is called (also with the time limit of 30 or 300 seconds). If DFPNS search disproves the position, we call Alpha-beta to find a move leading to the deepest loss.

The intermediate robot does not use DFPNS and it has only some features of Alpha-beta based algorithms (like the best evaluation function, but not the History Heuristic and the NegaScout). It searches only to the depth limited by two and half plies, i.e., five moves, and the random Alpha-beta is used in the whole game (the margin is again 20).

The beginner level is similar to the intermediate, but more dumb. It also does not use DFPNS and some Alpha-beta enhancements. Moreover it searches to the depth of only two plies, i.e., four moves, the margin for randomly selecting between the best moves is 5 000 and most importantly it has a very simple evaluation function which consists only of the incremental part of the best evaluation function (also with worse constants). Thus the evaluation is based mostly on the material part.

4. Tzaar Robot Results

In this chapter we show how our implementation of the algorithms described in Chapter 2 is successful. First we give results how the enhancements of the algorithms make our robot quicker and how we choose best values of constants used in the algorithms experimentally.

In the next section we show results against human opponents on Boiteajeux.net (BAJ) and against other robots that are available on the Internet and that are briefly described in Section 1.5. From the results we can conclude that the robot is able to play on the level of best players on BAJ server. We can also see that this program is now very likely the best available AI for Tzaar.

4.1 Experiments with the Robot

This section shows results of making the search as quick as possible.We make the search quicker by adding enhancements to the algorithms and setting parameters (constants) that were the best in experiments.

For parameter tuning and measuring the runtime duration we use two sets of Tzaar positions, one containing 19 middlegame positions for testing Alpha-beta and the other for testing DFPNS on endgame positions. All test positions were obtained from robot's games on BAJ.

The endgame positions are split into 43 positions that are hard for DFPNS, i.e., they cannot be solved within a minute and 15 seconds (with one exception), but a move after the position, DFNPS finds a solution quickly. The other 24 positions that are solved by DFPNS within one minute and not within only a second. Some positions in these sets are won for the player on turn, some are won for the other player. We remark that overall most positions are solved by DFPNS either in a time less than a second, or in time much more than one minute (which means practically that they cannot be solved by DFPNS).

Let us denote these test position sets by *middlegame positions*, *hard positions* and *solved positions*. They can be found on the attached CD.

In following sections we show tables with search durations in seconds for different enhancements or parameter settings for Alpha-beta and DFPNS. A time in a table is a sum of durations of searching each position in one of the test sets. In each section there are details about the search, e.g., which set of test positions was used. Note that when we test the influence of a constant on the search duration, other constants have their default values.

For automatically doing experiments, a *testing program* was used. For each position in the given set it starts the search and then reads the search duration from the output (without the time spent on loading a position and saving results of the search). The parameter or enhancement setting is compiled in the robot itself, but the algorithm used for the search and the time limit can be specified in command line parameters. The program outputs the sum of search durations of all positions in the set.

Tests were done on a server with processor Dual-Core AMD Opteron™ 2216 and approximately 64 GiB of memory, but only one of its cores and a small part

of memory were used[1]. The mistakes in the measurements are up to two seconds when the duration is about 80 seconds, but mostly they were less than a second.

4.1.1 Alpha-beta Enhancements

Table 4.1 shows how the enhancements for the Alpha-beta algorithm are successful. Constants used by the algorithms were already set appropriately according to experiments (see default values of constants in the next section).

Each middlegame position in the test set was run to depth of 3 plies, i.e., 6 moves. The depth of 6 moves was chosen, because it is often hard to search – the search duration for that depth is mostly above 30 seconds, but it only rarely exceeds two minutes, thus the depth is feasible for the robot. Depth 5 is usually done within a second in the middlegame and, on the other hand, depth 7 would last at least 30 times longer than depth six, thus it is mostly not feasible for our robot in the middlegame.

Enhancements	Duration (s)
All enhancements (ID, PV, MO, HH, NS)	825.1
Without NegaScout (ID, PV, MO, HH)	840.7
Without History Heuristic (ID, PV, MO, NS)	1687.1
Without NegaScout and History Heuristic (ID, PV, MO)	1157.3
Beginner's evaluation function with ID, PV and MO	1990.3
Only Principal Variation and Iterative Deepening	1803.0
Only Move Ordering and Iterative Deepening	3994.0
Only Iterative Deepening	3976.8
No enhancement (even without ID)	3841.3

Table 4.1: Duration of the Alpha-beta search with different enhancements.

From the table we can conclude that it is best to use all Alpha-beta enhancements, i.e., the Transposition Table with the Principal Variation (PV), the Iterative Deepening (ID), the Move Ordering (MO), the History Heuristic (HH) and the NegaScout (NS). We observe that the Principal Variation and the History Heuristic are the most important. Note also that the NegaScout without the History Heuristic is worse than not using the NegaScout.

The difference of over 800 seconds between the fourth and the fifth row is caused only by the fact that the beginner's evaluation function is much worse than the best one. All other tests in this section were done with the best evaluation function. The last two rows of the table show how much time the previous iterations of ID cost (in this case it is the search to depths from 1 to 5), since using only ID without PV is useful only for the time estimation.

4.1.2 Alpha-beta Enhancements Parameters

In this part we show how constants have importance on the duration of the Alpha-beta search. The only enhancements with parameters are the Move Ordering,

[1]This server also hosts our robot for the game server Boiteajeux.net.

the History Heuristic and the Transposition Table. The tests were done on the middlegame position set in the same way as in the previous section.

For the Move Ordering, there are three important constants in the formulas in Section 3.6: *SortStackBonus* (the default value is 19, Table 4.2), *SortStackCountMult* (the default value is 3, Table 4.3) and *SortCaptureCountMult* (the default value is 10, Table 4.4).

Bonus of stacking	Duration (s)
0	842.6
6	818.7
12	824.2
15	837.3
19	833.8
25	863.5

Table 4.2: Duration of the Alpha-beta search with different values of the constant *SortStackBonus*.

From Table 4.2 we observe that setting the bonus too high or too low, i.e., taking capture moves earlier or later in the order, increases the search duration. The default was 19 before the tests and 6 after them.

Multiplier of the count of a stacked piece type	Duration (s)
1	828.5
2	830.8
3	825.1
4	851.2
5	866.0

Table 4.3: Duration of the Alpha-beta search with different values of the constant *SortStackCountMult*.

Table 4.3 shows also that taking stacking moves later causes longer search duration, even those moves which make stacks of rare types of stones. From the different constant settings, we can observe the importance of the Move Ordering on the search duration.

From Table 4.4 we can conclude that it is better to try capture moves later than it was expected – the default value was 10 before the tests and 20 after them.

The History Heuristic is also involved in the Move Ordering. Durations for different values of a multiplier that determines the influence of HH in the Move Ordering formula for the first move of a player is shown in Table 4.5. Quite surprisingly, the best value is 20 and that is the same as the best value for the constant *SortCaptureCountMult*. The default value was 50 before the experiments and 20 after them.

Table 4.6 shows the influence of the size of the Transposition Table on the search duration. We observe that the size of $2^{16} = 65\,536$ is sufficient for searching to depth 6 and setting the size higher does not help much. The default value is 2^{19}.

Multiplier of the count of a captured piece type	Duration (s)
10	825.1
12	809.8
14	808.2
16	792.6
18	783.7
20	781.5
22	783.9
24	785.3
26	787.0
28	792.8
30	797.7
32	802.1

Table 4.4: Duration of the Alpha-beta search with different values of the constant $SortCaptureCountMult$.

Bonus of a stack when sorting	Duration (s)
3	801.4
5	782.0
10	793.9
15	778.6
20	773.9
25	781.9
30	782.7
40	783.1
50	825.1

Table 4.5: Duration of the Alpha-beta search with different values of the constant $sortHistoryPruneMult$.

Binary logarithm of the size of TT	Duration (s)
12	963.1
13	898.1
14	867.7
15	811.7
16	790.7
17	788.2
18	770.1
19	778.1
20	772.6
22	771.7
23	773.0
24	774.4

Table 4.6: Duration of the Alpha-beta search with different sizes of the Transposition Table.

4.1.3 DFPNS Enhancements

Table 4.7 shows the importance of the enhancements for DFPNS. Note that there is nearly no difference between the Dynamic Widening and Heuristic Weak DFPNS, also that sorting moves heuristically with the same algorithm as in Alphabeta is very useful and that one single enhancement is still not enough. The only solved position in the last four lines is still the same. The tests were done on solved endgame position set with the time limit of 60 seconds (often exceeded by 10 seconds due to overestimation of the time limit).

Enhancements	Solved (out of 24)	Duration (s)
DW, ET and EFB DFPNS	24	95.3
HW, ET and EFB DFPNS	24	94.6
HW, ET and EFB DFPNS without sorting moves	19	406.1
Only the Evaluation Function Based DFPNS	1	1718.6
Only the Heuristic Weak DFPNS	1	1712.9
Only the $1 + \epsilon$ Trick DFPNS	1	1734.4
DFPNS without enhancements	1	1716.1

Table 4.7: Results of DFPNS search with different enhancements.

4.1.4 DFPNS Enhancements Parameters

We start with a set of parameters set empirically or by doing some experiments, we adjust some of them and then we see that some positions from hard test position set are now solvable. The experiments are at first done on solved test positions, with Heuristic Weak PNS, the $1+\epsilon$ Trick and Evaluation Function Based DFPNS (unless otherwise stated) and again with the time limit of 60 seconds.

From Table 4.8 we observe that increasing the size of the Transposition Table does not lead to quicker solving of positions, but maybe some other positions can be solved with more memory. On the other hand, the size at least $2^{20} \doteq 1\,000\,000$ is needed – making TT smaller leads quickly to many unsolved positions.

Binary logarithm of the size of TT	Solved (out of 24)	Duration (s)
17	14	764.0
18	22	220.5
19	23	158.5
20	24	101.0
21	24	101.2
22	24	101.5
23	24	102.1
24	24	103.4
25	24	104.8

Table 4.8: Results of the DFPNS search with different sizes of the Transposition Table.

Now we try DFPNS on hard endgame positions with higher sizes of the Transposition Table. Table 4.9 shows that even the size of $2^{26} \doteq 64\,000\,000$ is not sufficient enough.

Binary logarithm of the size of TT	Solved (out of 43)	Duration (s)
20	1	3081.2
25	1	3125.7
26	1	3053.2

Table 4.9: Results of the DFPNS search on hard endgame positions with different sizes of the Transposition Table.

For the $1 + \epsilon$ Trick, only the value of constant ϵ is important. We modify the formula $1 + \epsilon$ to $1 + 1/EpsilonDivisor$ and Table 4.10 shows the search duration and the number of solved positions for different values of the constant $EpsilonDivisor$. Note that function defined by these values is not convex oppose to functions in the other tables. The default value is 8.

Divisor of the constant ϵ	Solved (out of 24)	Duration (s)
0.5	21	297.3
1	23	156.1
2	23	155.0
3	23	156.6
4	22	246.8
6	22	247.8
7	24	102.0
8	24	101.1
9	24	101.3
10	24	101.4
11	21	300.9
12	21	280.8
16	21	278.7

Table 4.10: Results of the DFPNS search with different values of the constant ϵ.

There are three important constants in Evaluation Function Based DFPNS called A, B (multipliers in the step function for proof and disproof numbers) and T (the threshold). Effect of the setting different values of these constants is shown in tables 4.11 (for A, the default value is 10), 4.12 (for B, the default value is 10) and 4.13 (for T, the default value is 1 000 000).

Quite surprisingly, the value of A can be anything within range from zero to approximately 40 without affecting the search duration and solvability (maybe with different values than 10 we can solve more positions by DFPNS).

From Table 4.12 we can conclude that constant B can be anything from four, and the search duration with the number of solved position are still the same (maybe we can solve more positions with some values of B).

As shown in Table 4.13, setting the threshold T for the step function in EFB DFPNS can improve the search duration with preserving the solvability. The constant T had default value 1 000 000 before these tests and after them 50 000 000 was chosen as the best value.

EFB DFPNS constant A	Solved (out of 24)	Duration (s)
0	24	100.8
1	24	101.1
2	24	100.8
4	24	103.7
6	24	101.6
8	24	101.9
10	24	101.0
20	24	101.4
30	24	101.0
40	24	104.1
50	22	213.9

Table 4.11: Results of the DFPNS search with different values of the constant A for EFB DFPNS.

EFB DFPNS constant B	Solved (out of 24)	Duration (s)
0	5	1426.8
1	17	562.3
2	22	231.8
4	24	101.8
8	24	101.4
10	24	101.0
20	24	102.3
30	24	101.3
40	24	101.3
50	24	101.0
100	24	101.4
500	24	101.4
5 000	24	102.2
500 000	24	101.6

Table 4.12: Results of the DFPNS search with different values of the constant B for EFB DFPNS.

Heuristic Weak DFPNS has two constants: the threshold T for the step function and the multiplier H. Table 4.14 shows that setting the threshold to nearly any value does not change neither the number of solved position, nor the search duration. The multiplier H can be 0, 1 or 2, but not more. We remark that the less H is, the algorithm behave more like Weak DFPNS without the step function. Also changing Heuristic Weak DFPNS to Weak DFPNS according to Hashimoto, Iida and Ueda [5] also leads to solving all positions within 94.7 seconds, thus we can conclude that Heuristic Weak DFPNS is equal to Weak DFPNS in the domain of Tzaar.

As we can see from Table 4.7 the Dynamic Widening has the same solvability and search duration as Weak DFPNS (both with the $1+\epsilon$ Trick and the Evaluation Function Based enhancement). Table 4.16 shows the search duration for different values of the constant J – disproof numbers only of J nodes in increasing order by

EFB DFPNS constant T	Solved (out of 24)	Duration (s)
100 000	3	1502.6
500 000	23	174.3
1 000 000	24	101.0
5 000 000	24	101.3
10 000 000	24	101.6
25 000 000	23	152.7
40 000 000	23	144.7
50 000 000	24	94.7
60 000 000	24	94.3
75 000 000	23	144.3
100 000 000	23	142.9
500 000 000	23	157.1

Table 4.13: Results of the DFPNS search with different values of the constant T for EFB DFPNS.

Heuristic Weak DFPNS constant T	Solved (out of 24)	Duration (s)
10	24	95.3
1 000	24	95.0
100 000	24	94.7
1 000 000	24	94.6
10 000 000	24	94.6
100 000 000	24	94.5
1 000 000 000	24	94.9

Table 4.14: Results of the DFPNS search with different values of the constant T for the Heuristic Weak DFPNS.

Heuristic Weak DFPNS constant H	Solved (out of 24)	Duration (s)
0	24	95.2
1	24	94.6
2	24	94.8
3	22	232.6
5	15	709.1

Table 4.15: Results of the DFPNS search with different values of the constant H for the Heuristic Weak DFPNS.

proof numbers are taken into account when counting the sum of disproof numbers. We observe that J can be anything in the range from 1 to approximately 25 without affecting solvability and with only a little change in the search duration.

With more appropriate constants we test DFPNS on hard test positions. With the time limit of 60 seconds, 21 out of 43 positions were solved within 1 628.5 seconds instead of only one solved position before experiments with constants. The number of solved positions remains the same when we set the time limit even to 600 seconds and also when we change Heuristic Weak DFPNS to the Dynamic Widening. Only byt setting the size of the TT to $2^{26} \doteq 64\,000\,000$ and the time limit to 600 seconds we get one more solved position, thus in whole 22.

The Dynamic Widening constant J	Solved (out of 24)	Duration (s)
1	24	94.6
2	24	94.3
5	24	95.3
10	24	95.7
15	24	96.8
20	24	98.1
25	24	98.1
30	23	165.6

Table 4.16: Results of the DFPNS search with different values of the constant J for the Dynamic Widening.

4.1.5 DFPNS versus Alpha-beta in Endgames

In this section we discuss whether to use Alpha-beta or DFPNS on endgame positions in Tzaar. Reasons why to use DFPNS are that a winning strategy can be quite long and a player can force his opponent to have only a little number of possible moves. On the other hand the branching factor is usually quite high even in the endgame (see Section 1.4) and it is hard to guess for DFPNS which move is worth trying.

Alpha-beta with all enhancements was tested on the solved and hard test position sets. Solving all solved positions (the easier ones for DFPNS) lasts 132.5 seconds, that is 40 seconds more than DFPNS. Moreover, the time limit needed to be set to 600 seconds, because in three cases the time estimation has terminated the search too early with the time limit of 60 seconds (but no search in fact lasts over 34 seconds). So sometimes it is better to use DFPNS, but the advantage over Alpha-beta is not very high.

But on the hard test set Alpha-beta solved 36 out of 43 positions within 3 220.4 seconds (with the time limit of 600 seconds, but it was often not used wholly) while DFPNS with better set constants solved only 22 positions.

Thus in the program we try to use DFPNS first (for a part of the time limit) and if it does not succeed because of the time limit, we call Alpha-beta. To deal with lost positions that were disproved by DFPNS we call Alpha-beta on them too, see Section 2.3 for theoretical details.

4.2 Playing with Other Programs and People

In this section we show how is our program successful against human and artificial opponents. We let the robot play on a game server against people and in a program against other programs for playing Tzaar.

4.2.1 Different Robot Levels against Each Other

To check whether the robot levels are set well, they played games against each other in the program called *Arena*. The program manages many games between two versions of the robot.[2] Each game starts in the fixed starting position and it

[2]It was also used for deciding which parameters are better for the evaluation function.

is random who is white.

The beginner robot was beaten by the intermediate robot 33 times and won 5 times, the intermediate robot was beaten by the expert robot 21 times and won 4 times and the expert robot was defeated by the unbeatable robot 19 times and won 6 times. From these results we conclude that the strength of robot's levels increases according to names of the levels.

4.2.2 Results with Other Computer Opponents

In this part we compare our program to other existing programs for automatic playing Tzaar that are available on the Internet (up to July 16, 2012). Except TZ1 [36] the comparison is based on at least few games between the program and our robot. TZ1 was not tested, since it does not play well as mentioned in Section 1.5. Also, it is written in Java and has to be modified to be used in some kind of an arena program.

Our robot played four game with hsTzaar [30], two as black and two as white, and won all of them. AI opponent in hsTzaar was set in the first part of the games to the second best level with number 2 and later to the best level 3, since level 3 is extremely slow in the beginning of a game – it can spend more than 10 minutes thinking about one move.

The four programs from the website of the University of Alaska [39] (Mockinator, Mockinator++, BiTzaarBot and GreensteinTzaarAI) are already implemented to play in the Daedalus Game Manager [25]. In this manager, similar to the Arena, four games between our robot and each of these programs were played (two games as white, two as black) and all of them were won by our robot.

Robots on BoardSpace.net [23] are particularly weak and they are not a challenge for an experienced player. After connecting our robot to the BoardSpace manager, only a few games were played and all has been won by our robot. Precisely, two games were against Smartbot, one playing as black and one playing as white, and two games were against Bestbot. Moreover, Bestbot thought for a long time, it was 9 minutes in the first game, and 18 minutes in the second game. Our robot needed only a few minutes (with the 30 second limit for searching a position) and Smartbot needed only a few minutes too. Since Dumbot is intended for beginners and plays worse than Smartbot and Bestbot, no test against it was done.

Up to my knowledge, we compare our robot to all available programs with AI for Tzaar, thus we can conclude that our robot is the best in playing Tzaar.

4.2.3 Results on Boiteajeux.net with Human Opponents

To test the program against people, the game server Boiteajeux.net (BAJ) was selected. On this server more than 30 games including Tzaar can be played in the way similar to *play by email* (a player send moves to his opponent by an email and then waits for the answer). Instead of emails, an HTML interface for each game is provided. For each game, an ELO rating is counted, i.e., for a win a player obtains some points according to his and opponent's ELO (it is always at least one point) and his opponent loses the same number of points. ELO of a new player is 1 500. Robots are on BAJ only for a few games and our robot is

the only one for Tzaar there.

There are two main reasons why to choose BAJ for the robot. The first is that robots communication with the server can be done by a simple HTTP client (it is checking games where the robot is on turn and playing moves). The second reason is that there are quite a lot people who play Tzaar on BAJ (between June 22 and July 5, 2012 over 50 Tzaar games were finished). The other possibility instead of BAJ is BoardSpace.net which run under Java, so one cannot create an HTTP client. Java robot would be also slow.

The robot in the expert level was released on March 20, 2012 under username *Pauliebot*. At first it contained a few bugs and corrected version was released on April 4, 2012. After that only minor updates were done, mostly improving the evaluation function. On April 24, 2012 the other levels of the robot were released (for the description of the levels see Section 3.7). New versions are announced on BAJ forum for Tzaar. To play with the robot, add the robot's username in the field Guests on the page for creating new games.

The beginner robot with username *PauliebotBeginner*, ELO 1 772 and 138 finished games is probably the most popular, because it is challenging even for intermediate players. For example rupelboom (ELO 1 931) won against it 14 times and lost 3 times, but tchako (ELO 1 452) who has 342 finished Tzaar games lost with the robot 8 times. Beginners, i.e., players with only a few finished Tzaar games, mostly do not win against the beginner robot. Thus the robot might have to play weaker, but I was not able to think out how to do it without making serious mistakes and still playing on the level of real beginners. The depth to which the robot searches is discussable.

The intermediate robot *PauliebotMedium* (ELO 1 616) played only 38 games and has ELO even lower than the beginner robot. The reason is maybe that only more experience players want to play with the intermediate robot instead of the beginner robot, thus they win more often. For example rupelboom (ELO 1 931) won all five games against the intermediate robot, PhilDakota (ELO 1852) won eight games and lost six games and surprisingly, tchako (ELO 1 452) won two times and lost four times. Thus the robot is challenging for intermediate players (ELO around 1 800), but it can be sometimes defeated by tchako who is losing with the beginner robot. The latter is probably caused by a different robot's strategy – tchako can defeat the intermediate's one, but not the beginner's one.

The unbeatable robot *PauliebotUnbeatable* has an inappropriate ELO 1 886 and thus many strong players do not want to play with it, because they would lose a lot of points from their ELO when they lose the game. To increase its ELO, games with other levels were played. The robot lost two games with Pauliebot and won also two games against it. Against the beginner and intermediate levels it won always with the exception of one game with the intermediate robot.

The unbeatable robot played only four games against people: one win and one loss against me (under username Paulie, ELO 2 173), one win against mat76 (ELO 1 597) and one loss against lynkowsky (ELO 1 851). Thus we can conclude that more time for the search and thus searching the game tree deeper with Alpha-beta does not help the robot to play much better. The reason is probably that the evaluation function was created for the lower time limit.

Up to July 30 the expert robot played so far 134 games (a few of these games were only for testing). It won 99 of them and it is the 13th best Tzaar player with

ELO 2 075. It played four times with the best Tzaar player on BAJ, SlowBrain (ELO 2 516), and won the second game which is a great success, since SlowBrain is far better than other players. With the second best Tzaar player, evrardmoloic (ELO 2 357), it won once and lost also once. With Gambit, the fourth player (ELO 2 207), it won three games and lost three games, but the first two lost games were before some mistakes in the program were corrected. Against me, playing under username Paulie (ELO 2 173, the 6th highest), the expert robot is mostly winning (while the unbeatable robot is winning against me only sometimes and I can defeat the intermediate and beginner robots quite easily). From these results we can conclude that the expert robot plays on the level of the best players on the server BAJ.

Some weaker players tried relatively many games against the expert robot. The most successful is Gregg (ELO 1 710) who won five out of 19 games and another is PhilDakota (ELO 1 852) who won three out of 17 games. So we can see that it is possible for intermediate players to defeat our robot despite the fact that the robot can won against the best players.

The reason why the expert robot lost some games on BAJ was often no stack of Tzarras, the second stone type in the number of pieces. We can observe that in two or three last turns of these lost games the robot had no chance to create a stack of Tzarras which could not be captured by the opponent. Before it the robot probably did not know that the opponent has such a trap. It is left to future work to improve our robot so that it cannot be defeated by this strategy.

Another thing that we can improve in robot's playing is preventing from losing medium stacks (size three or four) in the middlegame which happened in some games. From observations in Section 1.3, it can advantageous for the opponent to capture such stacks, although it is done by a double capture move and it mostly does not lead to robot's loss.

ELOs of players in this section were up to date July 30, 2012. The results can be found on Boiteajeux.net server using a search in the left menu.

Conclusion and Future Work

As we observed in Chapter 4 we created a robot that is able to play on the level of best players and that can defeat all other programs. We conclude that the Alpha-beta algorithm is quite successful in the domain of Tzaar and maybe in the domains of games that have large branching factor, and a good evaluation function is not so hard to implement (unlike Go).

We also tested the Depth-first Proof-number Search on endgame positions that have mostly still quite big number of possible moves. We found that performance of this algorithm is similar to Alpha-beta in endgames, but sometimes the Alpha-beta can solve positions in the endgame much quicker. This is caused by the high branching factor and the fact that there are not enough forcing moves in Tzaar.

We dealt with two problems specific for playing with people. First we proposed an algorithm based on Alpha-beta for randomly choosing a good enough move and second we discussed how to play in lost positions.

There are plenty of things left for future work and research. One can always adjust the evaluation function together with other constants in the program. Maybe there are positional or material properties that should be added to the evaluation function. Also the implementation of algorithms can be made better, for example we do not probably need to generate all moves every time, but we can maintain a list of possible moves for each player and modify it according to moves that are played. From the algorithmic view there are many Alpha-beta enhancements invented mostly for Chess that were not tested in the domain of Tzaar, e.g., maybe the Quiescence search can help.

DFPNS has performance similar to Alpha-beta in endgames, but it is left to future research to adjust DFPNS for games with a large branching factor, probably using some heuristics like the Move Ordering. The new algorithm Heuristic Weak PNS, developed from Weak PNS, turned out not to be successful for Tzaar, but in another domains such as Othello and Shogi, it may help.

Probably the most importantly we did not test the Monte Carlo Tree Search (MCTS) on Tzaar. This algorithm is very successful for Go, the best programs for Go use it, and Go has also a big branching factor as Tzaar. Hence MCTS is worth trying for Tzaar although in the endgame it can be better to try DFPNS or Alpha-beta that can solve the position deterministically.

Bibliography

[1] Allis, Victor. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis. University of Limburg, Maastricht, The Netherlands, 1994. ISBN 9090074880.

[2] Baudiš, Petr. *MCTS with Information Sharing*. Master thesis. Faculty of Mathematics and Physics, Charles University in Prague, 2011.

[3] Breuker, Dennis Michel, van den Herik, Hendrik Jacob, Uiterwijk, Jos W. H. M. *Replacement Schemes and Two-Level Tables*. ICCA Journal, 1996, pp. 175–180.

[4] Breuker, Dennis Michel. *Memory versus search in games*. Doctoral thesis. Universiteit Maastricht, 1998. ISBN 9090120068.

[5] Hashimoto, Junichi, Hashimoto, Tsuyoshi, Iida, Hiroyuki, Ueda, Toru. *Weak Proof-Number Search*. Proceedings of the 6th international conference on Computers and Games. Springer-Verlag, Berlin, Heidelberg, 2008, pp. 157–168. ISBN 978-3-540-87607-6.

[6] van den Herik, Hendrik Jacob, Winands, Mark. *Proof-Number Search and Its Variants*. Oppositional Concepts in Computational Intelligence. Springer, Berlin, Heidelberg, 2008, pp. 91–118. ISBN 978-3-540-70826-1.

[7] Heule, Marijn J. H., Rothkrantz, Leon J. M. *Solving games: Dependence of applicable solving procedures*. Science of Computer Programming, 2007, pp. 105–124. ISSN 0167-6423.

[8] Kishimoto, Akihiro, Müller, Martin, Yoshizoe, Kazuki. *Lambda depth-first proof number search and its application to go*. Proceedings of the 20th international joint conference on Artificial intelligence. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007, pp. 2404–2409.

[9] Kishimoto, Akihiro. *Dealing with Infinite Loops, Underestimation, and Overestimation of Depth-First Proof-Number Search*. AAAI Conference on Artificial Intelligence, 2010.

[10] Knuth, Donald Ervin, Moore, Ronald W. *An analysis of alpha-beta pruning*. Artificial Intelligence, 1975, pp. 293–326. ISSN 0004-3702.

[11] Kocsis, Levente, Szepesvári, Csaba. *Bandit Based Monte-Carlo Planning*. Machine Learning: ECML 2006. Springer, Berlin, Heidelberg, 2006, pp. 282–293. ISBN 978-3-540-45375-8.

[12] Lew, Łukasz, Pawlewicz, Jakub.*Improving depth-first PN-search:* $1 + \epsilon$ *trick*. Proceedings of the 5th international conference on Computers and games. Springer-Verlag, Berlin, Heidelberg, 2007, pp. 160–171. ISBN 3-540-75537-3, 978-3-540-75537-1.

[13] Nagai, Ayumu. *Df-pn algorithm for searching AND/OR trees and its applications*. Ph.d. thesis. The University of Tokyo, Tokyo, Japan, 2002.

[14] Reinefeld, Alexander. *An Improvement to the Scout Tree-Search Algorithm*. ICCA Journal, 1983, pp. 4–14.

[15] Schadd, Maarten, Winands, Mark. *Evaluation-Function Based Proof-Number Search*. Computers and Games. Springer, Berlin, Heidelberg, 2011, pp. 23–35. ISBN 978-3-642-17927-3.

[16] Schaeffer, Jonathan. *The History Heuristic*. Journal of the International Computer Chess Association, 1983, pp. 16–19.

[17] Schrüfer, Günther. *A strategic quiescence search*. ICCA Journal, 1989, pp. 3–9.

[18] Thomsen, Thomas. *Lambda-Search In Game Trees - With Application To Go*. Computers and Games. Springer, Berlin, Heidelberg, 2001, pp. 19–38. ISBN 978-3-540-43080-3.

[19] Wentink, Diederik. *Analysis and Implementation of the game Gipf*. M.Sc. thesis. Universiteit Maastricht, 2001.

[20] Yoshizoe, Kazuki. *A New Proof-Number Calculation Technique for Proof-Number Search*. Computers and Games. Springer, Berlin, Heidelberg, 2008, pp. 135–145. ISBN 978-3-540-87607-6.

[21] Zobrist, Albert Lindsey. *A new hashing method with application for game playing*. ICCA Journal, 1990, pp. 69–73.

[22] *TZAAR*. BoardGameGeek [online]. [cit. 2012-07-13]. URL: `http://boardgamegeek.com/boardgame/31999/tzaar`.

[23] *BoardSpace.net* [online]. [cit. 2012-07-13]. URL: `http://www.boardspace.net/`.

[24] *Boiteajeux.net* [online]. [cit. 2012-07-13]. URL: `http://www.boiteajeux.net/`.

[25] *Daedalus Game Manager*. Google Code [online]. [cit. 2012-07-13]. URL: `http://code.google.com/p/daedalus-game-manager/`.

[26] *GAMES Game Awards*. Games Magazin [online]. [cit. 2012-07-13]. URL: `http://www.gamesmagazine-online.com/gameslinks/archives.html#2009awards`.

[27] *GIPF project* [online]. [cit. 2012-07-13]. URL: `http://www.gipf.com/`.

[28] *2008 Golden Geek Best 2-Player Board Game Nominee*. BoardGameGeek [online]. [cit. 2012-07-13]. URL: `http://boardgamegeek.com/boardgamehonor/8763/2008-golden-geek-best-2-player-board-game-nominee`.

[29] *Project tzaar-ai*. Google Code [online]. [cit. 2012-07-30]. URL: `http://code.google.com/p/tzaar-ai/`.

[30] Vasconcelos, Pedro. *HsTZAAR* [online]. [cit. 2012-07-13]. URL: http://www.dcc.fc.up.pt/~pbv/stuff/hstzaar/.

[31] *2008 Nominees - General Strategy*. International Gamers Awards [online]. [cit. 2012-07-13]. URL: http://www.internationalgamersawards.net/winners-and-nominees/nominees/2008-nominees.

[32] *Nederlandse Spellenprijs - historie* [online]. [cit. 2012-07-13]. URL: http://www.spellenprijs.nl/historie.html.

[33] *Endgame Tablebases*. Chess Programming Wiki [online]. [cit. 2012-07-13]. URL: http://chessprogramming.wikispaces.com/Endgame+Tablebases.

[34] *Opening Book*. Chess Programming Wiki [online]. [cit. 2012-07-13]. URL: http://chessprogramming.wikispaces.com/Opening+Book.

[35] *Awarded Games 2008*. Spiel des Jahres [online]. [cit. 2012-07-13]. URL: http://www.spiel-des-jahres.com/cms/front_content.php?idart=925.

[36] Schwagereit, Johannes. *TZ1: A Program to play Tzaar* [online]. [cit. 2012-07-13]. URL: http://www.johannes-schwagereit.de/tz1/.

[37] *Tzaar Rules*. GIPF project [online]. [cit. 2012-07-13]. URL: http://www.gipf.com/tzaar/rules/rules.html.

[38] *Tzaar Strategy*. GIPF project [online]. [cit. 2012-07-13]. URL: http://www.gipf.com/tzaar/strategy/strategy.html.

[39] *Tzaar - AI Game Project for 2011*. Department of Mathematical Sciences, University of Alaska Anchorage [online]. [cit. 2012-07-13]. URL: http://www.math.uaa.alaska.edu/~afkjm/cs405/tzaar/index.html.

List of Figures

List of Tables

List of Abbreviations

- AI – Artificial Intelligence
- BAJ – Boiteajeux.net
- DFPNS – Depth-first Proof-number Search
- DGM – Daedalus Game Manager
- DN – Disproof number
- DW – Dynamic Widening
- EFB DFPNS – Evaluation Function Based Depth-first Proof-number Search
- ET – $1 + \epsilon$ Trick
- HH – History Heuristic
- HW – Heuristic Weak
- ID – Iterative Deepening
- MCTS – Monte Carlo Tree Search
- MO – Move Ordering
- NS – NegaScout
- MPN – Most Proving Node
- PNS – Proof-number Search
- PN – Proof number
- PV – Principal Variation
- TT – Transposition Table

A. Documentation of the Program

Our robot consists of the library tzaarlib for searching for best moves (described in Section A.3), the program tzaarmain that loads the board from a file and starts the search (Section A.2) and the Python web client for communication with Boiteajeux.net (Section A.1). For simple building tzaarlib and tzaarmain, a Makefile is provided. The current version of the source code is in the project tzaar-ai on Google Code [29].

On a CD attached to the thesis, there is the source code of our robot in the directory `robot`, the Python client for Boiteajeux.net (BAJ) in the directory `BAJclient`, test positions, used in Chapter 4, in the directory `testPositions`, and the Daedalus Game Manager [25] (DGM) in the directory `DaedalusGameManager` that can be used for offline playing with the robot. The source code of DGM is also provided, since it is modified to be used easier for offline playing with out robot.

A user documentation for offline playing with the robot using DGM can be found in the directory `DaedalusGameManager`. It is not written in this thesis, because DGM is not a part of our work and it might be improved or replaced by a user-friendlier program.

A.1 Python Web Client for Boiteajeux.net

Web client for communicating with the website Boiteajeux.net is written in Python. It downloads a page from BAJ and looks for a game in which the robot is on turn. When such a game is found the client downloads the page with board and calls the function `PlayGame`.

Function `PlayGame` has a parameter page containing HTML code of the page with board. From the HTML code it parses a position, converts it to the board representation (described in Section A.2.1) and saves it to a file. Then it calls the program tzaarmain and waits until the computation is done. Finally it loads best moves from a file and executes them on BAJ (via HTTP POST requests).

The client also looks for an invitation for playing – one can start a game with the robot by creating a new game and adding the robot's username in the field Guests. The client sends email to a given recipient when an exception occurs, and every day after midnight it sends statistics of searches done during the day.

Note that before using the client for BAJ, we have to configure it. That means, inside the Python code, fill the robot's username and password, an email where to send error messages and daily outputs, and directories where is the Tzaar robot located and where to save positions and outputs.

A.2 Program tzaarmain

The program tzaarmain written in C is quite simple. Given a file with a position in Tzaar it initializes arrays and variables with a board representation that are

in the library. Then it calls function `GetBestMove` in tzaarlib for searching for best moves and saves returned moves to a file.

The files and other options are given via command line arguments:

- `-a N` or `--ai N` – set the algorithm number `N` (see tzaarlib.h). The default AI number is in constant `MAINAI` in tzaarlib.h (it is AI used by the expert robot).
- `-b FILE` or `--bestmove=FILE` – search for best moves in a position stored in `FILE` and then save best moves into this file. An input file format is described in Section A.2.1 and output file format in Section A.2.2. This is a required option.
- `-e FILE` or `--execute=FILE` – execute best moves after searching for them and then save the position to `FILE`. Default is not to save any position after the search.
- `-t SECONDS` or `--timelimit=SECONDS` – set the time limit of the search to `SECONDS`. The default value is in the constant `AITIMELIMIT` (30 seconds).
- `-h` or `--help` – print usage.

A.2.1 Board Representation and Input Files

Now we describe a board representation and then the format of input and output files.

The Tzaar board is a hexagon with 5 fields on each side, but we cannot simply store it in the memory as a hexagon. Thus it is sloped to be fit in the quadratic array 9×9. In the library, one linear array of size 81 is used instead of a quadratic array.

Example of the array containing fixed starting position (array members are separated by spaces):

```
-1   1   1   1   1  100 100 100 100
-1  -2   2   2   2  -1  100 100 100
-1  -2  -3   3   3  -2  -1  100 100
-1  -2  -3  -1   1  -3  -2  -1  100
 1   2   3   1  100 -1  -3  -2  -1
100  1   2   3  -1   1   3   2   1
100 100  1   2  -3  -3   3   2   1
100 100 100  1  -2  -2  -2   2   1
100 100 100 100 -1  -1  -1  -1   1
```

The number 100 stands for a field outside the board and also for the field in the middle of the board. Empty fields with no stone have number 0. Other numbers stand for different types of stones:

- 1 is a white Tott,
- 2 is a white Tzarra,
- 3 is a white Tzaar,

- -1 is a black Tott,
- -2 is a black Tzarra,
- -3 is a black Tzaar.

In this array, a player can move in six directions: horizontally left, or right, vertically up, or down, and diagonally right and down, or left and up. The other diagonal directions (right and up, left and down) are not possible.

The heights of stacks are in another array of the same size. Fields outside the board and empty fields have stack height zero. Example for the fixed starting position:

```
1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 1 0
1 1 1 1 0 1 1 1 1
0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1
0 0 0 0 1 1 1 1 1
```

In the tzaarlib, there is also an array with names of fields (`"-"` is a field outside the board):

```
"A1", "B1", "C1", "D1", "E1", "-" , "-" , "-" , "-" ,
"A2", "B2", "C2", "D2", "E2", "F1", "-" , "-" , "-" ,
"A3", "B3", "C3", "D3", "E3", "F2", "G1", "-" , "-" ,
"A4", "B4", "C4", "D4", "E4", "F3", "G2", "H1", "-" ,
"A5", "B5", "C5", "D5", "-" , "F4", "G3", "H2", "I1",
"-" , "B6", "C6", "D6", "E5", "F5", "G4", "H3", "I2",
"-" , "-" , "C7", "D7", "E6", "F6", "G5", "H4", "I3",
"-" , "-" , "-" , "D8", "E7", "F7", "G6", "H5", "I4",
"-" , "-" , "-" , "-" , "E8", "F8", "G7", "H6", "I5"
```

The input file for the program is a sequence of numbers separated by some whitespace characters:

- first a player on turn is specified, i.e., robot's color. Number 1 stands for white, -1 for black,
- then there is the array with the board representation – 81 integers with types of stones, or 0 as an empty field, or 100 as a field outside the board,
- the last is the array with stack heights (81 nonnegative integers).

Example with comments after '%' (they should be deleted before using the file):

```
1 % our robot is white player

% stones on the board
 1  -1  -1  -1  -1  100 100 100 100
 1   2  -2  -2   2   3  100 100 100
 1   0  -3   0  -3   0   1  100 100
 0   0   0   0   0   0   2   1  100
 0  -3   0   0 100  0   3   0   1
100 -1   0  -3   0  -1  -3   2  -1
100 100 -2   0   0   1  -3  -2  -1
100 100 100  0   2   2   2  -2  -1
100 100 100 100  1   1   1   1  -1

% stack heights
1 1 1 1 1 0 0 0 0
1 1 1 1 1 3 0 0 0
1 0 1 0 1 0 1 0 0
0 0 0 0 0 0 1 1 0
0 3 0 0 0 0 1 0 1
0 1 0 1 0 1 1 1 1
0 0 3 0 0 2 1 1 1
0 0 0 0 1 1 1 1 1
0 0 0 0 1 1 1 1 1
```

Note that in the format it does not matter on whitespace characters (spaces, new lines and tabs). The file with numbers separated by a single space and no new lines will is loaded successfully.

A.2.2 Output File with Best Moves

Now we describe how best moves are saved in the output file. On the first line, there is the first move of a turn. It is saved as the name of the field from which a player moves a stack, and the name of the field where the player captured an opponent's stack (the names are separated by a space).

On the second line, there is an integer specifying what is the second move. Number -2 stands for no move (in the case of the first turn of white player, or win after the first move), -1 stands for a pass move, 0 for a stacking move, and 1 for a capture. In the last two cases, names of two fields follow, the first is the field from which a stack is moved, and the second is the field to which the stack is moved.

On the third line, there is some information about the search, namely the search duration in seconds (with three decimal places) and the returned value.

Example of a capture move and a stacking move:

```
G4 C6
0 F2 F1
33.303 33319
```

Example of a capture and a pass move (the returned value 2 000 000 000 means that the robot is going to win):

```
H2 D1
-1
2.742 2000000000
```

A.2.3 Module main

The module main contains function `main` which parses command line arguments using the getopt library and calls `ProcessPosition` which loads position. Then it calls `GetBestMove` in tzaarlib and finally it saves the result.

A.2.4 Module tzaarSaveLoad

The module tzaarSaveLoad contains functions for loading and saving positions in the format described in Section A.2.1 and a function for saving best moves in the format described in Section A.2.2. Function `SaveWholePosition` save all information about the position, including counts of stones according to a type, the zone of control, but this function is not used by the robot.

A.3 Library tzaarlib

The library tzaarlib contains the computation part of the program and it is also written in C (standard C99). For searching for best moves the algorithms Alpha-beta and Depth-first Proof-number Search (DFPNS) are implemented. There are also auxiliary functions for generating, executing and reverting moves. It is possible to choose between different versions of the algorithms, for example there are versions for beginners and intermediate players.

A.3.1 Arrays and Fields for Position Properties

For the current position representation there are arrays `board` and `stackHeights` of size 81 with the same format as described in Section A.2.1. The variable `player` is 1 when white is on turn and -1 when black is on turn. Since the turn of a player consists of two moves, the variable `moveNumber` determines which phase of a turn is: 1 is the first phase (a capture move) and 2 is the second (capture, stack, or pass).

In the variable `turnNumber`, there is the number of turns from the root position of the search (starting 1) – this is different from the number of turns in the whole game which is not known to the robot. Moves are stored in the array `history`.

There are some variables and arrays for storing information about the current position that can be counted directly from arrays `board` and `stackHeights`, but that would be very slow. The array `counts` contains the number of stacks alive (visible) for each stone type, the variable `stoneSum` is the sum of stacks on the board and it is used by the Iterative Deepening.

The variable `value` is the value of the current position determined by the evaluation function or by a search. The variable `materialValue` is counted by the part of the evaluation function that is counted incrementally when executing or reverting moves. The variable `hash` contains the value of the Zobrist hash function for the current position and it is also counted incrementally.

The array `highestStack` of the size equal to the number of stone types contains the height of the highest stack for each type. For incrementally maintaining this array library uses the quadratic array `countsByHeight` that contains the number of stones for each type and height.

The array `zoneOfControl` contains for each stone type how many stones of that type can be captured with one move. This is used in the evaluation function and for determining whether a player has any possible captures when he is on turn and it is his first move. The array is updated also incrementally using the array `threatenByCounts` of the size 81 which contains for each field how many stacks can capture a stack on this field.

A.3.2 Module tzaarlib

The module tzaarlib is the main module of the library. It contains definitions of structures, types, macros, global arrays and variables used throughout the library (some of them are described in the previous section).

The function `GetBestMove` starts the search according to the chosen algorithm and does the time estimation via the Iterative Deepening for the Alpha-beta based algorithms and for DFPNS via the estimation of the maximal number of nodes that can be searched.

In the header file, types and basic macros are defined first. Constants for properties of the game, types of AI (algorithms), and AI settings follows. The structure for a move, global variables, and arrays are defined at the end.

A.3.3 Module tzaarmoves

The module tzaarmoves contains functions for generating, executing and reverting moves. There are also helping functions for deallocating memory (free a single move or a linked list of moves), determining whether someone won in the current position, updating the zone of control after executing or reverting a move and converting between a field index and a field name (for example the field on index 3 has name D1).

Most of functions in this module are optimized to be as fast as possible, because they are called many times during the search. Note that the functions for generating moves are used for the first and the second move of a turn separately.

In the header file there are constants for the Move Ordering and arrays for possible directions that are used in the functions for generating moves.

A.3.4 Module tzaarinit

The module tzaarinit has functions for initializing arrays and variables with information about the current position. The counting of the hash value, the material value (the part of the value of a position that is counted incrementally during the search) and the zone of control is implemented here. Function `InitBoard` prepares starting position according to the parameter `setup` (random, or fixed) and calls other initialization functions, but it is not used by our robot.

Functions in this module are not optimized to be fast, because they are not called during the search, only before it. The values of the parameter `setup` and the fixed starting position are defined in the header file.

A.3.5 Module alphaBeta

The module alphaBeta contains functions that implement the Alpha-beta pruning algorithm with its enhancements, functions for working with the Transposition Table (TT) and static evaluation functions.

There are different Alpha-beta functions with different enhancements used:

- `AlphaBeta` – simple Alpha-beta with storing positions to TT,
- `AlphaBetaPV` – Alpha-beta with TT and the Principal Variation Move (PV),
- `AlphaBetaPVMO` – Alpha-beta with TT, PV and the heuristic Move Ordering (MO),
- `AlphaBetaMO` – Alpha-beta only with the Move Ordering (without TT),
- `AlphaBetaPVMORandom` – random Alpha-beta proposed in Section 2.1.3 with TT, PV and MO. It should be called only on the root of the search tree.
- `AlphaBetaPVMONegascout` – Alpha-beta with TT, PV, MO and Negascout,
- `AlphaBetaPVMOHistory` – Alpha-beta with TT, PV, MO and the History Heuristic,
- `AlphaBetaPVMOHistoryNegascout` – Alpha-beta with TT, PV, MO, the History Heuristic and the Negascout,
- `AlphaBetaPVMOBeginner` – Alpha-beta with TT, PV, MO and the beginner static evaluation function,
- `AlphaBetaPVMORandomBeginner` – random Alpha-beta with TT, PV, MO and the beginner static evaluation function. It should be called only on the root of the search tree.

The Alpha-beta enhancement Iterative Deepening is implemented in the module tzaarlib. The Transposition Table use the replacement scheme Two Big.

A.3.6 Module pns

The module pns contains the implementation of the Depth-first Proof-number Search (DFPNS) with some enhancements and the Transposition Table (TT) used by DFPNS (it differ from TT used by Alpha-beta).

There are different DFPNS functions with different enhancements used:

- `dfpns` – simple DFPNS without enhancements,
- `dfpnsEpsTrick` – DFPNS with the $1 + \epsilon$ Trick,
- `weakpns` – Heuristic Weak DFPNS (one can modify it to Weak DFPNS easily),
- `dfpnsEvalBased` – Evaluation Function Based DFPNS,
- `dfpnsWeakEpsEval` – Heuristic Weak DFPNS with the $1 + \epsilon$ Trick and the Evaluation Function Based enhancement,

- `dfpnsDynWideningEpsEval` – DFPNS with the Dynamic Widening, the $1+\epsilon$ Trick and Evaluation Function Based enhancement,

Constants for the enhancements are in the header file. The Transposition Table for DFPNS also use the scheme Two Big.

A.3.7 File hashedpositions.h

This header file contains data for the Zobrist hashing. There is a three dimensional array `HashedPositions` of unsigned 64bit integers (type thash) that contains random numbers for each combination of a field, a stone (or an empty field) and a stack height that can occur on the board. Impossible combinations have value zero.

The array is indexed in this way: `HashedPosition[field][stoneType][stackHeight]` where `field` is an index in the array `board` and it is in range from 0 to 80, `stoneType` is the value from the array `board` plus three (the value ranges from -3 to 3) and `stackHeight` is the height of a stack if there is any, or zero otherwise. Note that for fields outside the board, the values are not used.